# Strong Normalization in two Pure Pattern Type Systems

Benjamin Wack and Clément Houtmann[†]

*Université Henri Poincaré & LORIA[‡], FRANCE*

Benjamin.Wack@loria.fr

Clement.Houtmann@loria.fr

Pure Pattern Type Systems ($P^2TS$) combine in a unified setting the frameworks and capabilities of rewriting and $\lambda$-calculus. Their type systems, adapted from Barendregt's $\lambda$-cube, are especially interesting from a logical point of view. Strong normalization, an essential property for logical soundness, had only been conjectured so far: in this paper, we give a positive answer for the simply-typed system and the dependently-typed system. The proof is based on a translation of terms and types from $P^2TS$ into the $\lambda$-calculus. First, we deal with untyped terms, ensuring that reductions are faithfully mimicked in the $\lambda$-calculus. For this, we rely on an original encoding of the pattern matching capability of $P^2TS$ into the System $\mathsf{F}\omega$.
Then we show how to translate types: the expressive power of System $\mathsf{F}\omega$ is needed in order to fully reproduce the original typing judgments of $P^2TS$. We prove that the encoding is correct with respect to reductions and typing, and we conclude with the strong normalization of simply-typed $P^2TS$ terms. The strong normalization with dependent types is in turn obtained by an intermediate translation into simply-typed terms.

**Keywords:** Rewriting calculus, $\lambda$-calculus, Rewriting, Type systems, Strong normalization

[†] ENS de Cachan
[‡] UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

**Contents**

## 1. Introduction

The $\lambda$-calculus and term rewriting provide two fundamental computational paradigms that had a deep influence on the development of programming and specification languages, and on proof environments. The idea that having computational power at hand makes deduction significantly easier and safer is widely acknowledged (Dowek et al. 2003; Werner 1993). Starting from Klop's groundbreaking work on higher-order rewriting, and because of the complementarity of $\lambda$-calculus and term rewriting, many frameworks have been designed with a view to integrate these two formalisms.

This integration has been handled either by enriching first-order rewriting with higher-order capabilities or by adding algebraic features to the $\lambda$-calculus. In the first case, we find the works on CRS (Klop et al. 1993) and other higher-order rewrite systems (Nipkow and Prehofer 1998). In the second case, we can mention *case* expressions with dependent types (Coquand 1992), a typed pattern calculus (Kesner et al. 1996) and calculi of algebraic constructions (Blanqui 2001).

The *rewriting calculus*, *a.k.a.* $\rho$-calculus, by unifying the $\lambda$-calculus and the rewriting, makes all the basic ingredients of rewriting first-class citizens, in particular the notions of *rule application* and *result*. A rewrite rule becomes a first-class object which can be created, manipulated and customized in the calculus, whereas in works such as (Blanqui 2001; Blanqui et al. 2002), the rewriting system remains somehow external to the calculus.

In (Cirstea et al. 2001), a collection of type systems for the $\rho$-calculus was presented, extending Barendregt's $\lambda$-cube to a $\rho$-cube. These type systems have been studied deeper for $P^2TS$ (Barthe et al. 2003), a variant where the abstractors $\lambda$ and $\Pi$ have been distinguished whereas they were unified in the $\rho$-cube. The corresponding calculi have been proved to enjoy most of the usual good properties of typed calculi: substitution, subject reduction, uniqueness of types under certain assumptions, *etc.*

However, the rewriting calculus has also been assigned some type systems that do *not* prevent infinite reductions. These typed recursive terms are suitable for formally describing programs (especially in rule-based languages) and guaranteeing some safety properties. In particular, in (Cirstea et al. 2003), we have shown how to use them to encode the behavior of most term rewrite systems.

Conversely, $P^2TS$ have been designed for logical purposes. Therefore, in order to ensure consistency of the type systems, strong normalization is an important and desirable property, but it did remain an open problem. In this paper, we give a positive answer to this problem. Together with the consistency of normalizing terms, already proved in (Barthe et al. 2003), this result makes $P^2TS$ a good candidate for a proof-term language integrating deduction and computation at the same level.

The main contributions of this paper are:

— a clearer presentation of the type systems of $P^2TS$: with regard to the systems presented in (Barthe et al. 2003), a signature has been introduced in the typing judgments and some corrections have been made on product rules (in section 2 and 3);

— a compilation of pattern matching in the $\lambda$-calculus, which has other potential applications for the encoding of term rewriting systems (in section 5);

— a translation of the simply-typed system of $P^2TS$ to System $\mathsf{F}\omega$ emphasizing some particular typing mechanisms of $P^2TS$ (in section 6 and 7);
— a proof of strong normalization for simply-typed $P^2TS$ terms and for dependently-typed $P^2TS$ terms (in section 7 and 8).

This paper is organized as follows. In Section 2, we recall the syntax and the small-step semantics of $P^2TS$ and we state under which restrictions we work in this paper. In Section 3, we motivate and present the type systems of $P^2TS$. We recall some properties, and sketch the proof of strong normalization that will be developed in the rest of the paper. In Section 4, we recall the type system $\mathsf{F}\omega$ for the $\lambda$-calculus, into which we will translate $P^2TS$. In Section 5, we give an untyped version of the translation from $P^2TS$ to System $\mathsf{F}\omega$, showing how pattern matching is compiled and how reductions are preserved in the encoding. In Section 6, we emphasize the main difficulties of introducing types into the translation and we give the full translation, which we prove to be correct in Section 7. In Section 8, we give a translation from the dependently-typed system to the simply-typed system and we prove its correctness.

We assume the reader to be reasonably familiar with the notations and results of typed $\lambda$-calculi (Barendregt 1992), of the $\rho$-calculus (Cirstea et al. 2003) and of $P^2TS$ (Barthe et al. 2003). This paper is a revised and extended version of (Wack 2004), presented at TCS'2004.

## 2. $P^2TS$: dynamic semantics

In this section, we recall the syntax of $P^2TS$ and their evaluation rules.

**Notations** To denote a tuple of terms $B_k \dots B_n$, we will use the vector notation $\overline{B}_{(k..n)}$, or simply $\overline{B}$ when $k$ and $n$ are obvious from the context. This notation will be used in combination with operators according to their default associativity: for instance, $A\overline{B} \triangleq AB_1 \dots B_n$ and $\lambda\overline{P{:}\Delta}.A \triangleq \lambda P_1{:}\Delta_1 \dots \lambda P_n{:}\Delta_n.A$. The vector notation will also be used for substitutions.

Most usually, $x, y, z$ will denote *variables*; $A, B$ are *terms*; $P, Q$ are *patterns*; $a, f, g$ are *constants*; $s, s_1, s_2$ are special constants called *sorts*; $C, D$ are *types* (which are in fact terms too); $\iota$ is an *atomic type*; $\Gamma, \Delta$ are *contexts* (mainly in $P^2TS$); $\Sigma$ is a *signature*. All of those are defined by the grammar below. Moreover, we will use $\alpha, \alpha_f$ for an *arity*, and $\theta$ for a *substitution*, which will be defined a bit later.

**The calculus** The syntax of $P^2TS$ extends that of the typed $\lambda$-calculus with structures and patterns (Barthe et al. 2003).

| | |
|---|---|
| *Signatures* | $\Sigma ::= \emptyset \mid \Sigma, f{:}A$ |
| *Contexts* | $\Gamma ::= \emptyset \mid \Gamma, x{:}A$ |
| *Patterns* | $P ::= x \mid f\,\overline{P}$ |
| *Terms* | $A ::= s \mid f \mid x \mid \lambda P{:}\Delta.A \mid \Pi P{:}\Delta.A \mid [P \ll_\Delta A]A \mid A\,A \mid A \wr A$ |

A term with shape $\lambda P{:}\Delta.A$ is an *abstraction* with pattern $P$, body $A$ and context $\Delta$. The term $[P \ll_\Delta B]A$ is a *delayed matching constraint* with pattern $P$, body $A$, argument $B$ and context $\Delta$. In an *application* $A\,B$, the term $A$ represents the function, while the term $B$ represents the argument. The application of a constant symbol, say $f$, to a term

$A$ will also be denoted by $f\,A$; it follows that the usual algebraic notation of a term is currified, *e.g.* $f(A_1,\ldots,A_n) \triangleq f\,A_1\,\cdots\,A_n = f\,\overline{A}$. A term $(A \wr B)$ is called a *structure* with elements $A$ and $B$, roughly denoting a set of terms. A term $\Pi P{:}\Delta.A$ is a *dependent product*, and will be used as a type.

**Remark 1 (Legal patterns).** Several choices could be made for the set of patterns $P$: in this paper, *we only consider algebraic patterns*, whose shape is defined above. With respect to the original patterns of $P^2TS$, we introduce two restrictions which will be applied throughout he whole paper:

— We forbid patterns featuring $\lambda$-abstractions (which is in fact enforced by the given grammar) because, from a logical point of view, they amount to defining new constants in the signature. Indeed, with the matching algorithm given in (Barthe et al. 2003), the free variables of a pattern (for instance $y$ in $\lambda x.xy$) can be instantiated only with subterms of the argument that do not feature any bound variable of the pattern (for instance this pattern matches only terms of shape $\lambda x.xA$ with $x \notin \mathcal{FV}(A)$). Therefore, $A$ must have the same type as $y$ and must be typable in the same context as the whole term $\lambda x.xA$. Thus, solving the matching problem $\lambda x.xy \ll \lambda x.xA$ is equivalent to solving $fy \ll fA$ where $f$ is a constant with a convenient type.

— In every binding construct (*i.e.* $\lambda P{:}\Delta.A$ and $\Pi P{:}\Delta.A$ and $[P \ll_\Delta A]A$), we impose $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$ (which will be enforced by the typing system), so that no variable can be instantiated in the pattern of an abstraction, because a term such as $(\lambda x : (x{:}\Phi).\lambda x{:}\emptyset.B)A$ (featuring only algebraic patterns) would reduce to $\lambda A.B$, which may be no longer in our term grammar.

We recall the notion of free variables for $P^2TS$.

**Definition 1 (Domain of a context/signature).**
The domain of a context is defined as follows:

$$\mathcal{D}om(\emptyset) = \emptyset \qquad \mathcal{D}om(\Gamma, x{:}A) = \mathcal{D}om(\Gamma) \cup \{x\}$$

The domain of a signature is defined similarly.

**Definition 2 (Free variables).**
The set of free variables $\mathcal{FV}$ of a term is inductively defined as follows:

$$\begin{array}{lcllcl}
\mathcal{FV}(A \wr B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{FV}(x) & \triangleq & \{x\} \\
\mathcal{FV}(A\,B) & \triangleq & \mathcal{FV}(A) \cup \mathcal{FV}(B) & \mathcal{FV}(f) & \triangleq & \emptyset \\
\mathcal{FV}(\lambda P{:}\Delta.A) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(\Delta)) \setminus \mathcal{D}om(\Delta) & & & \\
\mathcal{FV}(\Pi P{:}\Delta.A) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(\Delta)) \setminus \mathcal{D}om(\Delta) & & & \\
\mathcal{FV}([P \ll_\Delta B]A) & \triangleq & (\mathcal{FV}(A) \cup \mathcal{FV}(\Delta) \cup \mathcal{FV}(B)) \setminus \mathcal{D}om(\Delta) & & & \\
\mathcal{FV}(\Gamma, x{:}A) & \triangleq & \mathcal{FV}(\Gamma) \cup \mathcal{FV}(A) & & &
\end{array}$$

In this paper, extending Church's notation, the context $\Delta$ in $\lambda P{:}\Delta.B$ (resp. $[P \ll_\Delta B]A$ or $\Pi P{:}\Delta.B$) contains the type declarations of the free variables appearing in the pattern $P$, *i.e.* $\mathcal{D}om(\Delta) = \mathcal{FV}(P)$. These variables are bound in the (pattern and body of the) abstraction. When the pattern is just a variable $x$, we may abbreviate $\lambda(x : (x{:}A)).B$

| | | | | |
|---|---|---|---|---|
| $(\rho)$ | $(\lambda P{:}\Delta.A)\,B$ | $\to_\rho$ | $A\theta_{(P \ll B)}$ | if $\theta_{(P \ll B)}$ exists |
| $(\sigma)$ | $[P \ll_\Delta B]A$ | $\to_\sigma$ | $A\theta_{(P \ll B)}$ | if $\theta_{(P \ll B)}$ exists |
| $(\delta)$ | $(A \wr B)\,C$ | $\to_\delta$ | $A\,C \wr B\,C$ | |

Fig. 1. Top-level rules of $P^2TS$.

(resp. $\Pi(x : (x{:}A)).B$) into the usual notation $\lambda x{:}A.B$ (resp. $\Pi x{:}A.B$). The context $\Delta$ will be omitted when we consider untyped terms.

As usual, we work modulo $\alpha$-conversion and we adopt Barendregt's *hygiene-convention* (Barendregt 1992), *i.e.* free and bound variables have different names. Equality of terms modulo $\alpha$-conversion will be denoted by $\equiv$. This allows us to define substitutions properly:

**Definition 3 (Substitution).** A (finite) substitution $\theta$ is a function from the set of variables to terms which differs from the identity only on a finite set. Its application to $x$ is denoted $x\theta$. If for all $i \in [1..n], x_i\theta \equiv A_i$ and $\theta$ is the identity everywhere else, $\theta$ has domain $\mathcal{D}om(\theta) = x_1 \dots x_n$ and we will also write it $[x_1 := A_1 \dots x_n := A_n]$.

The application of the substitution $\theta = [x_1 := A_1 \dots x_n := A_n]$ to a term $B$ (denoted $B\theta$) is defined below. By $\alpha$-conversion of $B$, we can assume that no bound variable of $B$ belongs to $\mathcal{D}om(\theta) \cup \bigcup_{i=1}^{n} \mathcal{F}V(A_i)$.

$$
\begin{array}{lcllcl}
f\theta & \triangleq & f & (\lambda P{:}\Delta.B)\theta & \triangleq & \lambda P{:}(\Delta\theta).(B\theta) \\
x_i\theta & \triangleq & \begin{cases} A_i & \text{if } x_i \in Dom(\theta) \\ x_i & \text{otherwise} \end{cases} & (\Pi P{:}\Delta.B)\theta & \triangleq & \Pi P{:}(\Delta\theta).(B\theta) \\
 & & & (A \wr B)\theta & \triangleq & (A\theta) \wr (B\theta) \\
(AB)\theta & \triangleq & (A\theta)(B\theta) & ([P \ll_\Delta B]A)\theta & \triangleq & [P \ll_{\Delta\theta} B\theta](A\theta) \\
\emptyset\theta & \triangleq & \emptyset & (\Delta, x{:}A)\theta & \triangleq & \Delta\theta, x{:}(A\theta)
\end{array}
$$

$P^2TS$ features pattern abstractions whose application requires solving matching problems, which we will denote as $P \ll A$. For the purpose of this paper, we consider only syntactic matching, since it can be described with a quite simple algorithm and yields at most one solution (which will be denoted $\theta_{(P \ll A)}$ if it exists). It ensures confluence of the reduction we will define further without a particular evaluation strategy. The only difficulty when proving this result arises when considering non-linear patterns, since we should check equality of two terms.

The top-level rules are presented in Fig. 1. By the $(\rho)$ rule, the application of a term $\lambda P{:}\Delta.A$ to a term $B$ consists in solving the matching equation $P \ll B$ and applying the obtained substitution (if it exists) to the the term $A$; the $(\sigma)$ rule does the same for $[P \ll_\Delta B]A$, which allows to bind variable and perform pattern matching in the types. If no solution exists, the rules $(\rho)$ and $(\sigma)$ are not fired and the corresponding terms are not reduced. The $(\delta)$ rule distributes structures on the left-hand side of the application. This gives the possibility to apply "in parallel" two distinct abstractions $A$ and $B$ to a term $C$. The relation $\mapsto_{\rho\sigma\delta}$ is defined as the congruent closure of $\to_\rho \cup \to_\sigma \cup \to_\delta$, and $\twoheadmapsto_{\rho\sigma\delta}$ (resp. $=_{\rho\sigma\delta}$) denotes the reflexive and transitive (resp. reflexive, symmetric and transitive) closure of $\mapsto_{\rho\sigma\delta}$.

**Proposition 1 (Confluence (Barthe et al. 2003)).** With algebraic patterns and syntactic matching, $P^2TS$ are confluent under $\mapsto_{\beta\sigma}$.

## 3. $P^2TS$: static semantics

In (Cirstea and Kirchner 2000), a simple type system was introduced for the rewriting calculus (an alternative formulation of $P^2TS$); however, it allows to prove strong normalization of terms only at the price of a strong restriction over the types of constants. In the next subsection, we explain why some types had to be forbidden.

In the second subsection, we present a version of the type systems of $P^2TS$. In this framework, any type is allowed for a constant, but we use a richer type system integrating patterns into types, reminiscent of a dependent types discipline.

### 3.1. *Naive simple types*

The example of this section is taken from (Cirstea et al. 2003). Let us consider a quite straightforward system of simple types for $P^2TS$: given an atomic type $\iota$, types (noted $\sigma$ or $\tau$) are described by the grammar

$$\sigma \quad ::= \quad \iota \mid \sigma \twoheadrightarrow \sigma$$

When typing an abstraction, we just replace the type of the abstracted variable with the type of the whole pattern :

$$\frac{\Gamma, \Delta \vdash P : \sigma \qquad \Gamma, \Delta \vdash A : \tau}{\Gamma \vdash \lambda P{:}\Delta.A : \sigma \twoheadrightarrow \tau} \qquad \frac{\Gamma \vdash A : \sigma \twoheadrightarrow \tau \qquad \Gamma \vdash B : \sigma}{\Gamma \vdash A\,B : \tau}$$

Such a type system enjoys some good properties such as subject reduction, uniqueness and decidability of typing, *etc.* On the other hand, it allows one to typecheck also terms with infinite reductions (we ommit type annotations for readability since they do not play a role in reductions).

Let $f : (\iota \twoheadrightarrow \iota) \twoheadrightarrow \iota$ and $\omega \triangleq \lambda(f\,x).\big(x\,(f\,x)\big)$. Then:

$$\begin{array}{rcl} \omega\,(f\,\omega) & \triangleq & \big(\lambda(f\,x)\,.\,(x\,(f\,x))\big)\,(f\,\omega) \\ & \mapsto_{\rho} & [f\,x \ll f\,\omega](x\,(f\,x)) \\ & \mapsto_{\sigma} & \omega\,(f\,\omega) \\ & \mapsto_{\rho} & \ldots \end{array}$$

Still, the term is typable; let $\pi_1$ be the following derivation :

$$\frac{x : \iota \twoheadrightarrow \iota \vdash f : (\iota \twoheadrightarrow \iota) \twoheadrightarrow \iota \qquad x : \iota \twoheadrightarrow \iota \vdash x : \iota \twoheadrightarrow \iota}{x : \iota \twoheadrightarrow \iota \vdash f\,x : \iota}$$

and $\pi_2$ be the following one :

$$\dfrac{\dfrac{\pi_1}{x : \iota \twoheadrightarrow \iota \vdash f(x) : \iota} \qquad \dfrac{\dfrac{}{x : \iota \twoheadrightarrow \iota \vdash x : \iota \twoheadrightarrow \iota} \qquad \dfrac{\pi_1}{x : \iota \twoheadrightarrow \iota \vdash f\, x : \iota}}{x : \iota \twoheadrightarrow \iota \vdash x\,(f\, x) : \iota}}{\vdash \lambda(f\, x).(x\,(f\, x)) : \iota \twoheadrightarrow \iota}$$

Then we can conclude :

$$\dfrac{\dfrac{\pi_2}{\vdash \omega : \iota \twoheadrightarrow \iota} \qquad \dfrac{\dfrac{}{\vdash f : (\iota \twoheadrightarrow \iota) \twoheadrightarrow \iota} \qquad \dfrac{\pi_2}{\vdash \omega : \iota \twoheadrightarrow \iota}}{\vdash f\, \omega : \iota}}{\vdash \omega\,(f\, \omega) : \iota}$$

Therefore, this type system is not appropriate for using $P^2TS$ as a proof-term language: cut elimination would not hold, and the corresponding logic could even be proved unsound. One may think that this is due to the occurrence of constants whose type corresponds to an unprovable proposition. However, the example above can also be carried on with every occurrence of $\iota$ replaced by $\iota \twoheadrightarrow \iota$, and then the type of $f$ is a provable proposition.

### 3.2. *The Pure Pattern Type Systems*

The type systems of (Cirstea et al. 2001; Barthe et al. 2003) were designed in order to provide a strongly normalizing calculus where there is no restriction on the type of the constants (except from those enforced by the type system). Until now, strong normalization was an open problem for all these systems.

The system we study here is a slight variant of (Barthe et al. 2003). The inference rules are given in Fig. 2. For a more explicit manipulation of constants, we have introduced a signature $\Sigma$ which, like in the Edinburgh Logical Framework, prevents the type of a constant to depend on free variables. The judgments $\Sigma$ *sig* describe what a legal signature is, and the judgments $\vdash_\Sigma$ describe what legal typed terms are given a signature $\Sigma$.

Like in traditional Pure Type Systems, the system is conditioned by three sets: a set $\mathcal{S}$ of sorts; a set of pairs of sorts $\mathcal{A}$ describing what the legit axioms are; a set of pairs of sorts $\mathcal{R}$ describing what the legit product rules are (in fully general PTS, this last contains triplets instead of pairs, but we directly apply a usual restriction since the systems we focus on in this paper are not the most general). We discuss here the main modifications with respect to traditional Pure Type Systems.

— The (Struct) rule says that a structure $A \wr B$ can be typed with type $C$ if $A : C$ and $B : C$, hence forcing all members of the structure to be of the same type;
— The (Abs) rule deals with $\lambda$-abstractions in which we bind over patterns. By means of the well-formedness of the product type, it requires in particular that the pattern and body of the abstraction are typable in the extended context $\Gamma, \Delta$;
— The (Appl) rule, which deals with applications, imposes that the resulting type in the

$$(\textsc{Sig})\ \frac{}{\emptyset\ sig} \qquad (\textsc{WeakΣ})\ \frac{\Sigma\ sig \quad \vdash_\Sigma A : s \quad f \notin \mathcal{D}om(\Sigma)}{\Sigma, f{:}A\ sig}\ (s \in \mathcal{S})$$

$$(\textsc{Axiom})\ \frac{\Sigma\ sig}{\vdash_\Sigma s_1 : s_2}\ ((s_1,s_2) \in \mathcal{A}) \qquad (\textsc{Const})\ \frac{\Sigma\ sig \quad f{:}A \in \Sigma}{\vdash_\Sigma f : A}$$

$$(\textsc{Var})\ \frac{\Gamma \vdash_\Sigma A : s}{\Gamma, x{:}A \vdash_\Sigma x : A}\ (x \notin \Gamma, s \in \mathcal{S}) \qquad (\textsc{Struct})\ \frac{\Gamma \vdash_\Sigma A : C \qquad \Gamma \vdash_\Sigma B : C}{\Gamma \vdash_\Sigma A \wr B : C}$$

$$(\textsc{WeakΓ})\ \frac{\Gamma \vdash_\Sigma A : B \quad \Gamma \vdash_\Sigma C : s \quad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}C \vdash_\Sigma A : B}\ (x \notin \Gamma, s \in \mathcal{S})$$

$$(\textsc{Conv})\ \frac{\Gamma \vdash_\Sigma A : B \qquad \Gamma \vdash_\Sigma C : s}{\Gamma \vdash_\Sigma A : C}\ \begin{pmatrix} s \in \mathcal{S} \\ B =_{\beta\delta} C \end{pmatrix}$$

$$(\textsc{Abs})\ \frac{\Gamma, \Delta \vdash_\Sigma B : C \qquad \Gamma \vdash_\Sigma \Pi P{:}\Delta.C : s}{\Gamma \vdash_\Sigma \lambda P{:}\Delta.B : \Pi P{:}\Delta.C}\ \begin{pmatrix} \mathcal{D}om(\Delta) = \mathcal{F}V(P) \\ (s \in \mathcal{S}) \end{pmatrix}$$

$$(\textsc{Appl})\ \frac{\Gamma \vdash_\Sigma A : \Pi P{:}\Delta.C \qquad \Gamma \vdash_\Sigma [P \ll_\Delta B]C : s}{\Gamma \vdash_\Sigma A\,B : [P \ll_\Delta B]C}\ (s \in \mathcal{S})$$

$$(\textsc{Prod})\ \frac{\Gamma, \Delta \vdash_\Sigma P : A \qquad \Gamma, \Delta \vdash_\Sigma A : s_1 \qquad \Gamma, \Delta \vdash_\Sigma C : s_2}{\Gamma \vdash_\Sigma \Pi P{:}\Delta.C : s_2}\ \begin{pmatrix} \mathcal{D}om(\Delta) = \mathcal{F}V(P) \\ (s_1,s_2) \in \mathcal{R} \end{pmatrix}$$

$$(\textsc{Match})\ \frac{\Gamma, \Delta \vdash_\Sigma P : A \quad \Gamma \vdash_\Sigma B : A \quad \Gamma, \Delta \vdash_\Sigma A : s_1 \quad \Gamma, \Delta \vdash_\Sigma C : s_2}{\Gamma \vdash_\Sigma [P \ll_\Delta B]C : s_2}\ \begin{pmatrix} \mathcal{D}om(\Delta) = \mathcal{F}V(P) \\ (s_1,s_2) \in \mathcal{R} \end{pmatrix}$$

Fig. 2. The typing rules of $P^2TS$.

conclusion features delayed matching. In case the delayed matching can be successfully solved, one can recover the expected type by applying the conversion rule;

— The rules (Match) and (Prod) regulate the formation of product types. They ensure that the pattern and the body of the product are typable in the extended context $\Gamma, \Delta$. They are parametrized with pairs of sorts which can be either $*$ (for terms) or $\Box$ (for types). The legal pairs are given in a set $\mathcal{R}$. A given choice of $s_1$ and $s_2$ then leads to a type system where one can build "$s_2$ depending on $s_1$". For instance, every system includes the product rule $(s_1, s_2) = (*, *)$ so that terms depending on terms (*i.e.* functions) can be built.

**Remark 2 (Case-dependent structures).**

The rule (Struct) is rather restrictive, since one can not build case-dependent expressions such as $\lambda 0.0 \wr \lambda(s\,x).x$ because of the distinct patterns appearing in the types $\Pi 0{:}\emptyset.nat$ and $\Pi(s\,x){:}(x{:}nat).nat$. However, it is non-trivial to find a less strict rule such that elaborated structures can be built without losing strong normalization. An example of this phenomenon is developed in Subsection 3.4.

**Theorem 1 (Properties of the typed calculi).** (Cirstea et al. 2002; Barthe et al. 2003)

1 Substitution
If $\Gamma, x : C, \Delta \vdash_\Sigma A : D$ and $\Gamma \vdash_\Sigma B : C$, then $\Gamma, \Delta[x := B] \vdash_\Sigma A[x := B] : D[x := B]$.
2 Subject reduction
If $\Gamma \vdash_\Sigma A : C$ and $A \mapsto\!\!\!\!\!\rightarrow_{\beta\delta} A'$, then $\Gamma \vdash_\Sigma A' : C$.
3 Uniqueness of types up to second order
If $\mathcal{R} \subseteq \{(*, *), (\Box, *)\}$ (*i.e.* if only terms depending on terms and terms depending on types are allowed), then :
if $\Gamma \vdash_\Sigma A : C_1$ and $\Gamma \vdash_\Sigma A : C_2$, then $C_1 =_{\beta\delta} C_2$.

### 3.3. *Specific properties of $\rho_\rightarrow$*

In this paper, we mainly treat the case of the simply typed calculus $\rho_\rightarrow$, corresponding to $\mathcal{R} = \{(*, *)\}$. Thus, in all the remainder except Section 8, the sorts $s_1, s_2$ appearing in (Match) and (Prod) rules are always $*$. The sort $\Box$ is still required for the typing of $*$ itself, which is useful for type constants such as $\iota : *$. In particular, this implies uniqueness of types.

Let us prove some additional properties peculiar to the system $\rho_\rightarrow$, needed for the translation into $\lambda$-calculus.

**Lemma 1.**
In the system $\rho_\rightarrow$, if $\Gamma \vdash_\Sigma A : \Box$ then $A \equiv *$.

*Proof.* By induction on a derivation for the judgment $\Gamma \vdash_\Sigma A : \Box$, distinguishing over the last rule we use. □

**Definition 4 (Types).** In the system $\rho_\rightarrow$ and for a context $\Gamma$, we will call a *type* any term $C$ such that $\Gamma \vdash_\Sigma C : *$ and such that there is no structure in $C$.

**Lemma 2 (Shape of types).**
Types belong to the following language:

$$C \ ::= \ x \mid \iota \mid \Pi P{:}\Delta.C \mid [P \ll_\Delta A]C$$

and for any type $C$

— the type variables $x$ appearing in $C$ are such that $\Gamma, \overline{\Delta} \vdash_\Sigma x : *$, where $\overline{\Delta}$ denotes contexts appearing in the $\Pi$-abstractions and in the matching constraints in $C$;
— the atomic types $\iota$ appearing in $C$ are constants in $\Sigma$ such that $\vdash_\Sigma \iota : *$.

*Proof.* By induction on a derivation for the judgment $\Gamma \vdash_\Sigma C : *$, distinguishing over the last rule we use. Notice that the conversion rule is useless, since $\Gamma \vdash_\Sigma C : *$ and $*$ can not be converted to any other typable term in $\rho_\rightarrow$ (this other term would have type $\Box$ too, and by Lemma 1 it must be $*$). □

Notice that a type variable $x$ can not be abstracted in $\rho_\rightarrow$, which makes it roughly equivalent to a constant. Therefore, we can assume there is no type variable in a type.

Let us recall that there is no arity attached to the constants of $\Sigma$; however, the types allow us to recover this notion to a certain extent. The following lemma makes precise an intuitive property of $\rho_{\rightarrow}$: that a term can be applied to as many arguments as there are $\Pi$-abstractions in its type.

**Definition 5 (Maximal arity).**
We define the *maximal arity* $\alpha$ of a type by:

$$\begin{array}{rcl} \alpha(x) & \triangleq & 0 \\ \alpha(\iota) & \triangleq & 0 \\ \alpha(\Pi P{:}\Delta.C) & \triangleq & 1+\alpha(C) \\ \alpha([P \ll_\Delta A]C) & \triangleq & \alpha(C) \end{array}$$

**Lemma 3.**
Let $\Gamma$ be a fixed context. In $\rho_{\rightarrow}$, for any term $A$ such that $\Gamma \vdash_\Sigma A : C$, if $\Gamma \vdash_\Sigma A\,\overline{B}_{(1..k)} : D$, then $k \leq \alpha(C)$ and $\alpha(D) = \alpha(C) - k$.

*Proof.* It is easy to check that $\alpha$ is stable by type conversion, since the variable $x$ occurring at the rightmost position in the type can not be instantiated (that would require at least the rule $(\Box, *)$).

We proceed by induction on $k$.

**if** $k = 0$ **:** trivial

**if** $0 < k$ **:** the term $A\,\overline{B}_{(1..k)}$ being typable, so are its subterms. Hence

$$\Gamma \vdash_\Sigma A\,\overline{B}_{(1..k-1)} : E$$

for some $E$. By induction hypothesis, we have $k-1 \leq \alpha(C)$ and $\alpha(E) = \alpha(C)-(k-1)$. It is easy to see that a derivation for $A\,\overline{B}_{(1..k)}$ must use the rule

$$(\textsc{Appl})\ \frac{\Gamma \vdash_\Sigma A\,\overline{B}_{(1..k-1)} : \Pi Q{:}\Delta.E_1 \qquad \Gamma \vdash_\Sigma [Q \ll_\Delta B_k]E_1 : s}{\Gamma \vdash_\Sigma A\,\overline{B}_{(1..k)} : [Q \ll_\Delta B_k]E_1}$$

where $E =_{\beta\delta} \Pi Q{:}\Delta.E_1$ and $D =_{\beta\delta} [Q \ll_\Delta B_k]E_1$. We can now conclude that $\alpha(E) = \alpha(E_1) + 1 \geq 1$ hence $\alpha(C) = \alpha(E) + k - 1 \geq k$ and also that $\alpha(D) = \alpha(E_1) = \alpha(E) + 1 = \alpha(C) - (k-1) + 1 = \alpha(C) - k$.

□

In particular, we will use this notion for constants and structures:

— if $\vdash_\Sigma f : C$ we note $\alpha_f$ the integer $\alpha(C)$ and by abuse of language we can call it maximal arity of $f$ (since the signature $\Sigma$ can be considered as fixed beforehand, and any other type of $f$ is convertible to $C$ and thus has the same arity);
— if $\Gamma \vdash_\Sigma A \wr B : C$ we note $\alpha_{A \wr B}$ the integer $\alpha(C)$.

**Corollary 1 (Compatibility of arities in a matching equation).**
If a redex $(\lambda(f\,\overline{P}_{(1..p)}).A)\,(g\,\overline{B}_{(1..q)})$ or $[f\,\overline{P}_{(1..p)} \ll g\,\overline{B}_{(1..q)}]A$ appears in a term typable in $\rho_{\rightarrow}$, then $\alpha_f - p = \alpha_g - q$.

*Proof.* Indeed, the rule (Match) must be used in any type derivation for such a term,

and its premises enforce that $\Gamma \vdash_\Sigma f\,\overline{P}_{(1..p)} : A$ and $\Gamma \vdash_\Sigma g\,\overline{B}_{(1..q)} : A\theta$. The shape of types in $\rho_\to$ ensures that $\alpha(A) = \alpha(A\theta)$, so immediately $\alpha_f - p = \alpha(A) = \alpha_g - q$.

In particular, when $f \equiv g$, the condition $\alpha_f - p = \alpha_g - q$ reduces to $p = q$, which is essential for solving such a matching equation. □

### 3.4. *About strong normalization*

Let us take back the example from Subsection 3.1 and try to typecheck it. With product types, the type of the constant $f$ should be given as follows:

$$\Sigma \;\equiv\; \iota : *,\; f : \Pi(y \,:\, (\Pi(z : \iota).\iota))\,.\,\iota$$

Let $\Delta \equiv x : \Pi(z{:}\iota).\iota$, we have then:

$$\vdash_\Sigma \lambda(f\,x){:}\Delta.\big(x\,(f\,x)\big) \;:\; \Pi(f\,x){:}\Delta.\iota$$

with $\omega \triangleq \lambda(f\,x){:}\Delta.\big(x\,(f\,x)\big)$. So, to form the term $f\,\omega$, we should use the rule (Appl) and then use (Match) for the second premise. This leads us into proving that the pattern $y$ (appearing in the type of $f$) has the same type as $\omega$, which is not possible: $y$ has type $\Pi z{:}\iota.\iota$ whereas $\omega$ has type $\Pi(f\,x){:}\Delta.\iota$, and those two are not convertible because of their distinct patterns.

As mentionned before, a more general typng rule for structures would open the door again to such non-normalizing terms. For instance, one could imagine a rule keeping only a "most general" pattern in the types, yielding:

$$\frac{\lambda 0.0 \;:\; \Pi 0{:}\emptyset.nat \qquad \lambda(s\,x){:}(x{:}nat).x \;:\; \Pi(s\,x){:}(x{:}nat).nat}{\vdash \lambda 0.0 \wr \lambda(s\,x).x \;:\; \Pi n{:}nat.nat}$$

However, with such a rule, the term we studied in the previous subsection can be typed again with a little modification: $\omega' \triangleq \lambda y{:}\alpha.y \wr \lambda(f\,x){:}\Delta.\big(x\,(f\,x)\big)$ where $\Delta$ stands for $x : \Pi(z{:}\alpha).\alpha$ and $\Sigma$ for $f : \Pi x{:}\Delta.\alpha$.

$$\frac{\vdash_\Sigma \lambda y{:}\alpha.y \;:\; \Pi y{:}\alpha.\alpha \qquad \vdash_\Sigma \lambda(f\,x){:}\Delta.\big(x\,(f\,x)\big) \;:\; \Pi(f\,x){:}\Delta.\alpha}{\vdash_\Sigma \;\lambda y{:}\alpha.y \wr \lambda(f\,x){:}\Delta.\big(x\,(f\,x)\big) \;:\; \Pi w{:}\alpha.\alpha}$$

Then $\vdash f\,\omega' : \alpha$ and $\vdash \omega'\,(f\,\omega') : \alpha$ and we have the infinite reduction:

$$\begin{aligned}
\omega'\,(f\,\omega') &\triangleq \big(\lambda y.y \wr \lambda(f\,x)\,.\,(x\,(f\,x))\big)\,(f\,\omega') \\
&\mapsto_\delta (\lambda y.y)\,(f\,\omega') \wr (\lambda(f\,x)\,.\,(x\,(f\,x)))\,(f\,\omega') \\
&\mapsto_\beta\mapsto_\sigma f\,\omega' \wr \omega'\,(f\,\omega') \\
&\mapsto\!\!\!\!\to_{\beta\sigma\delta} f\,\omega' \wr f\,\omega' \wr \omega'\,(f\,\omega') \\
&\mapsto\!\!\!\!\to_{\beta\sigma\delta} \dots
\end{aligned}$$

A promising research direction for relaxing the (Struct) rule without losing normalization is to use intersection types, keeping *both* patterns in the types and carefully using matching failures to simplify the intersection type when possible.

We have just seen that one cannot typecheck $f\,\omega$ (and thus $\omega\,(f\,\omega)$) in the type system of Fig. 2. In fact, this property holds for any not strongly normalizing term:

**Theorem 2 (Strong normalization of typable $P^2TS$ terms).**
For all $\Sigma, \Gamma, A, C$, if $\Gamma \vdash_\Sigma A : C$ in $\rho_\to$ then $A$ is strongly normalizing.

The remaining of the paper (except Section 8) is devoted to the proof of this theorem. In Section 5, we will build an encoding function $[\![\ ]\!]$ from $P^2TS$ into $\lambda$-calculus such that if $A$ has an infinite reduction, then $[\![ A ]\!]$ has an infinite reduction. In Section 6, we will enrich this translation so that if $A$ is typable, then $[\![ A ]\!]$ is typable in System $\mathsf{F}\omega$. The strong normalization of $A$ is then a consequence of the strong normalization theorem for System $\mathsf{F}\omega$.

As a conclusion to this paragraph, let us briefly explain why usual reducibility techniques seem to fail for this typed calculus. Roughly speaking, the interpretation of a type $\Pi P{:}\Delta.C$ should be a function space whose domain is defined not only as the interpretation of the type of $P$ but also as terms matching with $P$ and whose suitable subterms belong to the interpretations of the types appearing in $\Delta$. This imbrication of interpretations leads to circularities in the definitions of interpretations. Of course, our translation would allow us to derive adequate but unnatural reducibility candidates from those existing for System $\mathsf{F}\omega$; the direct definition of proper candidates based on $P^2TS$ types remains an open problem.

## 4. The System $\mathsf{F}\omega$

Our encoding will produce $\lambda$-terms typable in the type system $\mathsf{F}\omega$, first introduced and studied in (Girard 1972). This formalism and its properties have been generalized to the Calculus of Constructions (Coquand and Huet 1988), and later on to Pure Type Systems. We follow here the generic presentation of (Barendregt 1992). The terms, types and kinds are taken in the following set:

$$Pseudo{-}terms \qquad t ::= * \mid \Box \mid x \mid \lambda x{:}t.t \mid \Pi x{:}t.t \mid t\,t$$

The inference rules describing the legal terms are given in Fig. 3. Here, the possible product rules $(s_1, s_2)$ are $\{(*,*),(\Box,*),(\Box,\Box)\}$. The one-step $\beta$-reduction

$$(\lambda x{:}t.u)\,v \mapsto_\beta u[x := v]$$

will be denoted $\mapsto_\beta$; its reflexive and transitive closure will be denoted $\twoheadmapsto_\beta$; its symmetric, reflexive and transitive closure will be denoted $=_\beta$.

**Notations** By convention, objects belonging to the $\lambda$-calculus will be denoted with lower case letters. In a given context $\Gamma$, a pseudo-term $t$ is said to be:

— a *kind* (denoted $k$) if $\Gamma \vdash_{\mathsf{F}\omega} t : \Box$.
— a *type* (denoted $\sigma, \tau$) if there is a kind $k$ such that $\Gamma \vdash_{\mathsf{F}\omega} t : k$. Type variables are denoted $\beta, \gamma$.
— a *term* (denoted $t, u$) if there is a type $\tau$ such that $\Gamma \vdash_{\mathsf{F}\omega} t : \tau$. Term variables are denoted $w, x, y, z$.

The unicity of typing (Theorem 3.3) guarantees the non-overlapping of these three sets of terms.

These rules use only $\Pi$-abstractions of shape $\Pi x{:}\sigma.\tau$. However, if $x \notin \mathcal{FV}(\tau)$, the usual type arrow $\sigma \to \tau$ can be used as an abbreviation. We will use it in both following cases:

$$\frac{}{\vdash_{\mathsf{F}\omega} * : \Box}\ (\textsc{axiom}) \qquad \frac{\Gamma \vdash_{\mathsf{F}\omega} \sigma : s \qquad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}\sigma \vdash_{\mathsf{F}\omega} x : \sigma}\ (\textsc{var})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} t : \sigma \qquad \Gamma \vdash_{\mathsf{F}\omega} \tau : s \qquad x \notin \mathcal{D}om(\Gamma)}{\Gamma, x{:}\tau \vdash_{\mathsf{F}\omega} t : \sigma}\ (\textsc{weak})$$

$$\frac{\Gamma, x : \sigma \vdash_{\mathsf{F}\omega} t : \tau \qquad \Gamma \vdash_{\mathsf{F}\omega} \Pi x{:}\sigma.\tau : s}{\Gamma \vdash_{\mathsf{F}\omega} \lambda x{:}\sigma.t : \Pi x{:}\sigma.\tau}\ (\textsc{abs})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} t : \Pi x{:}\sigma.\tau \qquad \Gamma \vdash_{\mathsf{F}\omega} u : \sigma}{\Gamma \vdash_{\mathsf{F}\omega} t\, u : \tau[x := u]}\ (\textsc{appl})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} t : \tau \qquad \Gamma \vdash_{\mathsf{F}\omega} \sigma : s \qquad \sigma =_\beta \tau}{\Gamma \vdash_{\mathsf{F}\omega} t : \sigma}\ (\textsc{conv})$$

$$\frac{\Gamma \vdash_{\mathsf{F}\omega} \sigma : s_1 \quad \Gamma, x{:}\sigma \vdash_{\mathsf{F}\omega} \tau : s_2 \quad (s_1, s_2) \in \{(*,*), (\Box,*), (\Box,\Box)\}}{\Gamma \vdash_{\mathsf{F}\omega} \Pi x{:}\sigma.\tau : s_2}\ (\textsc{prod})$$

Fig. 3. The typing rules of $\mathsf{F}\omega$.

— if $\vdash_{\mathsf{F}\omega} \sigma : *$, then $\vdash_{\mathsf{F}\omega} \tau : *$ since we do not consider dependent types in the $\lambda$-calculus. For the same reason, $x$ does not appear in $\tau$ — it would require the product rule $(*, \Box)$.
— if $\vdash_{\mathsf{F}\omega} \sigma : \Box$ and $\vdash_{\mathsf{F}\omega} \tau : \Box$, *i.e.* when $\Pi x{:}\sigma.\tau$ is a kind, we also know that $x$ can not appear in $\tau$. It is essentially for the same reason as before: we would need a product rule $(\Box, \triangle)$ with $\vdash_{\mathsf{F}\omega} \Box : \triangle$.

**Theorem 3 (Properties of System $\mathsf{F}\omega$).** (Girard 1972; Barendregt 1992)

1 Substitution
If $\Gamma, x : \sigma, \Delta \vdash_{\mathsf{F}\omega} t : \tau$ and $\Gamma \vdash_{\mathsf{F}\omega} u : \sigma$, then $\Gamma, \Delta[x := u] \vdash_{\mathsf{F}\omega} t[x := u] : \tau[x := u]$.
2 Subject reduction
If $\Gamma \vdash_{\mathsf{F}\omega} t : \sigma$ and $t \mapsto\!\!\!\!\rightarrow_\beta t'$, then $\Gamma \vdash_{\mathsf{F}\omega} t' : \sigma$.
3 Unicity of typing
If $\Gamma \vdash_{\mathsf{F}\omega} t : \Box$ then $t$ has no other type than $\Box$.
If $\Gamma \vdash_{\mathsf{F}\omega} t : k$ and $\Gamma \vdash_{\mathsf{F}\omega} t : k'$, then $k \equiv k'$.
If $\Gamma \vdash_{\mathsf{F}\omega} t : \sigma$ and $\Gamma \vdash_{\mathsf{F}\omega} t : \tau$, then $\sigma{=_\beta}\tau$.
4 Strong normalization
If $\Gamma \vdash_{\mathsf{F}\omega} t : \sigma$, then $t$ has no infinite reduction.

## 5. Untyped translation

In this section we translate (into the $\lambda$-calculus) the untyped $\rho_{\rightarrow}$-terms with algebraic (possibly non-linear) patterns and syntactic matching.

If we were to consider more elaborated matching theories, in most cases we should choose an evaluation strategy for the rewriting calculus to be confluent, and it is this particular strategy that we would be encoding, losing generality. Moreover, we should

enrich our compilation in order to generate all the possible solutions of a matching problem: for instance, associative matching can generate an arbitrarily high number of distinct solutions. We conjecture that most *syntactic equational theories* (as defined in (Kirchner and Klay 1990)) could be encoded, since they can be described by applying one rule at the head of a term and then only considering subterms, which should be feasible with typed $\lambda$-terms.

The process of syntactic matching consists in discriminating whether the argument begins with the expected constant, and recursively use pattern matching on subterms. It is this simple algorithm that we compile into the $\lambda$-calculus (without even checking for the equality of the head constants). Non-linear patterns are linearized during the compilation, so the equality tests that we should perform on subterms corresponding to a non-linear variable are simply discarded. Therefore, our compilation is complete but incorrect: it may yield a solution for a matching equation that is not solvable. However, since we want to preserve the length of reduction, completeness is the only property we are interested in.

For this encoding to work, we need to assume that each constant (or structure) is given with a maximal arity (as defined in Lemma 3 and fulfilling Corollary 1). As shown in Subsection 3.3, this notion of maximal arity is strongly linked to the existence of a typed framework and a signature, but it could be adapted to some untyped situations too: for instance, if we were to encode the repeated application of a Term Rewriting System, structures would be used in a very restrictive way, and the arity of the constants would be given.

The translation is given in Fig. 4, by a recursive function $[\![\cdot]\!]$ mapping $P^2TS$-terms to $\lambda$-terms. Since we are talking about untyped terms here, we do not treat the cases of $\Pi$-abstractions, matching constraints and sorts, neither do we annotate abstractions with contexts. Every variable we introduce must be fresh; they will be all bound in the $\lambda$-term, except for the variable $x_\perp$ which will remain free. We use it essentially for representing some matching failures, and as a wildcard for some useless arguments of $\lambda$-terms. Recall that $\alpha_f$ (resp. $\alpha_{A \wr B}$) denotes the maximal arity of $f$ (resp. $A \wr B$).

$$
\begin{aligned}
[\![ x ]\!] &\triangleq x \\
[\![ f ]\!] &\triangleq \lambda \overline{x}_{(1..\alpha_f)}.\ \lambda z.(z\, \overline{x}_{(1..\alpha_f)}) \\
[\![ A \wr B ]\!] &\triangleq \lambda \overline{x}_{(1..\alpha_{A \wr B})}.\ \big((\lambda z.[\![ A ]\!] \overline{x}_{(1..\alpha_{A \wr B})})([\![ B ]\!] \overline{x}_{(1..\alpha_{A \wr B})})\big) \\
[\![ \lambda x.A ]\!] &\triangleq \lambda x.[\![ A ]\!] \\
[\![ \lambda(f\, \overline{P}_{(1..p)}).A ]\!] &\triangleq \lambda u.(u \overline{x_\perp}_{(p+1..\alpha_f)}\, [\![ \lambda \overline{P}_{(1..p)}.\lambda \overline{x'}_{(p+1..\alpha_f)}.A ]\!]) \\
&\quad \text{(renaming the possible multiple occurrences of a variable in } \overline{P}_{(1..p)}) \\
[\![ A\, B ]\!] &\triangleq [\![ A ]\!]\, [\![ B ]\!]
\end{aligned}
$$

Fig. 4. Untyped term translation

**Remark 3.** The special case of symbols with $\alpha_i = 0$ gives:

$$\begin{array}{rcl} [\![ f ]\!] & \triangleq & \lambda z.z \\ [\![ \lambda f.A ]\!] & \triangleq & \lambda u.(u[\![ A ]\!]) \end{array}$$

Let us explain this translation:

— In $[\![ f ]\!]$, the variables $x_1 \dots x_{\alpha_f}$ will be instantiated by the arguments $\overline{B}$ of $f$ (which explains why we had to bound the arity of $f$). The variable $z$ can be instantiated by any function which must fetch the different arguments of $f$, which allows the simulation of matching.

— $[\![ A \wr B ]\!]$ is translated into a $\lambda$-term which embeds both $[\![ A ]\!]$ and $[\![ B ]\!]$, with the abstractions needed to distribute some eventual arguments to both translated subterms. The encoding differs from the usual pair of the $\lambda$-calculus in the sense that it can reduce to (an $\eta$-expansion of) $[\![ A ]\!]$ only, but it is not a concern for proving strong normalization. At the level of types, it will make the translation easier.

— In $[\![ \lambda x.A ]\!]$, the abstraction over a single variable is straightforwardly translated into a $\lambda$-abstraction.

— In $[\![ \lambda(f\,\overline{P}_{(1..p)}).A ]\!]$, the variable $u$ takes the argument of this function (for instance $g\,\overline{B}_{(1..q)}$) and applies it to various parameters. If necessary, the $\alpha_f - p$ occurrences of the variable $x_\perp$ instantiate the remaining variables $x_{q+1} \dots x_{\alpha_g}$ which can appear in $[\![ g ]\!]$: this is where we use the condition $\alpha_f - p = \alpha_g - q$. Then the last argument instantiates $z$ in $[\![ g ]\!]$, so that each argument of $g$ is matched against the corresponding subpattern. The equality of the head constants is not checked.
The fresh variables $x'_{p+1} \dots x'_{\alpha_f}$ will be instantiated by the $\alpha_f - p$ first $x_\perp$'s and do not modify the remaining reductions since they do not appear in $[\![ A ]\!]$. If a variable $x$ has multiple occurrences in the pattern, by $\alpha$-conversion, only one of the subpatterns $P_i$ will get the "original" variable, and the other $x$'s are renamed to fresh variables not occurring in $[\![ A ]\!]$. The correctness of this choice will be proved in Theorem 4.

— $[\![ A\,B ]\!]$ just consists in applying the translation of one term to the translation of the other.

**Example 1 (Translation of a term).**
Let $a$ and $f$ be two constants with $\alpha_a = 0$ and $\alpha_f = 1$. The term $\big(\lambda y.(\lambda(f\,x).x)\,y\big)\,(f\,a)$ (types of variables are omitted) is translated as follows:

$$\underbrace{(\lambda y.\Big(\overbrace{(\lambda u.(u\,(\lambda x.x)))}^{[\![\lambda(f\,x).x]\!]}\,y\Big))}_{[\![\lambda y.(\lambda(f\,x).x)\,y]\!]}\Big(\overbrace{(\lambda x_1.\lambda z.(z x_1))}^{[\![ f ]\!]}\overbrace{(\lambda v.v)}^{[\![ a ]\!]}\Big)$$

**Example 2 (Translation of a $\rho$-reduction).**
The only reduction path from the term in example 1 is:

$$(\lambda y.(\lambda(f\,x).x)\,y)\,(f\,a) \quad \mapsto_p \quad (\lambda(f\,x).x)\,(f\,a) \quad \mapsto_p \quad a$$

In particular the internal "redex" $(\lambda(f\,x).x)\,y$ can be reduced only when $y$ has been instantiated. Let's see how the translated term mimics this behavior: even if we reduce

first the $\beta$-redexes in the subterm $[\![\,(\lambda(f\,x).x)\,y\,]\!]$, we see that the instantiation of $y$ is mandatory here too in order to end the reduction.

At each reduction step, the selected $\lambda$-abstraction and its argument are underlined.

$$\begin{array}{rl}
 & (\lambda y.\Big((\underline{\lambda u}.(u(\lambda x.x)))\underline{y}\Big))\Big((\lambda x_1.\lambda z.(z x_1))(\lambda v.v)\Big) \\
\mapsto_\beta & (\lambda y.(y(\lambda x.x)))\Big((\underline{\lambda x_1}.\lambda z.(z x_1))\underline{(\lambda v.v)}\Big) \\
\mapsto_\beta & (\underline{\lambda y}.(y(\lambda x.x)))\underline{(\lambda z.(z(\lambda v.v)))} \\
\mapsto_\beta & (\underline{\lambda z}.(z(\lambda v.v)))\underline{(\lambda x.x)} \\
\mapsto_\beta & (\underline{\lambda x}.x)\underline{(\lambda v.v)} \\
\mapsto_\beta & (\lambda v.v) \\
= & [\![\,a\,]\!]
\end{array}$$

Let us consider now a matching failure.

**Example 3 (Translation of a matching failure).**

Let us take $\Sigma = \{g{:}\Pi x{:}(x{:}\iota).\iota, f{:}\Pi x{:}(x{:}\iota).\iota\}$, hence the maximal arities $\alpha_f = \alpha_g = 1$. The term $(\lambda(f\,x).x)\,(g\,y)$ is in normal form since the head constants $f$ and $g$ differ. Still, as our translation does not take into account the head constants, this matching failure is not reproduced in the translation.

$$\begin{array}{rl}
 & \big(\overbrace{\lambda u.(u(\lambda x.x))}^{[\![\,\lambda(f\,x).x\,]\!]}\big)\big(\overbrace{(\underline{\lambda x_1}.\lambda z.(z x_1))}^{[\![\,g\,]\!]}\,\underline{y}\big) \\
\mapsto_\beta & (\underline{\lambda u}.(u(\lambda x.x)))\underline{(\lambda z.(z y))} \\
\mapsto_\beta & (\underline{\lambda z}.(z y))\underline{(\lambda x.x)} \\
\mapsto_\beta & (\underline{\lambda x}.x)\underline{y} \\
\mapsto_\beta & y
\end{array}$$

We end this section by proving that the encoding preserves the reductions of the initial $\rho$-term.

**Lemma 4 (Closure by substitution).**

For any $\rho$-terms $A$ and $B_1, \ldots, B_n$, for any variables $x_1, \ldots x_n$,

$$[\![\,A[x_1 := B_1 \ldots x_n := B_n]\,]\!] \quad = \quad [\![\,A\,]\!][x_1 := [\![\,B_1\,]\!] \ldots x_n := [\![\,B_n\,]\!]] \tag{1}$$

*Proof.* It is easy to see that free (resp. bound) variables remain free (resp. bound) in the translation, and that $x_\perp$ is the only new free variable.

We proceed by two nested inductions : one on the structure of $A$, then another on he structure of the pattern (in the case where $A$ is an abstraction). Variables are translated as themselves and constants are not affected by the substitutions, hence the base cases are correct.

In all the inductive cases except for the abstraction, the translation of $A$ uses directly the translation of all the subterms of $A$, hence the substitution is propagated directly to those translated subterms.

The case of abstraction is treated by another induction over the pattern $P$: we show that, if $A$ fulfills equation (1), then for any set of patterns $P_1 \dots P_m$, the abstraction $\lambda\overline{P}_{(1..m)}.A$ fulfills it too.

The main point here is to check that the translation function $[\![\ ]\!]$ evaluates in a finite number of recursive calls: patterns are not affected by the substitution, so the propagation of the property (1) during the induction is trivial.

The order on sets of patterns $\overline{P}_{(1..m)}$ is defined by the following measure:

$$\begin{aligned} \#(\overline{P}_{(1..m)}) &= \sum_{j=1}^{m} \#(P_j) \\ \#(x) &= 1 \\ \#(f\overline{P}_{(1..p)}) &= \sum_{j=1}^{p} \#(P_j) + \alpha_f + 1 \end{aligned}$$

Let us check that this measure decreases during the translation. If the pattern $P_1$ is only a variable then the translation rule for $[\![ \lambda x.A ]\!] = \lambda x.[\![ A ]\!]$ applies, and the measure $\#$ decreases by 1.

Otherwise, we have a term with shape $[\![ \lambda f\overline{Q}_{(1..p)}.\lambda P_2 \dots \lambda P_m.A ]\!]$. The translation decomposes the first pattern $f\overline{Q}_{(1..p)}$ into its sub-patterns and at most $\alpha_f$ abstractions $\lambda x'$, and the constant $f$ disappears. The measure then decreases by at least 1:

$$\#(\overline{Q}_{(1..p)}\,\overline{x'}_{(p+1..\alpha_f)}) \;\leq\; \#(\overline{Q}_{(1..p)}) + \alpha_f \;=\; \#(f\overline{Q}_{(1..p)}) - 1$$

□

We will use the same order to treat the case of abstractions in the proof of Theorem 4.

**Theorem 4 (Faithful reductions).**
For any terms $A$ and $B$, if $A \mapsto_{\beta\delta\rho} B$, then $[\![ A ]\!] \twoheadrightarrow_{\beta} [\![ B ]\!]$ in at least one step.

*Proof.* Again by induction on the structure of $A$.

As for the closure by substitution lemma, if the reduction occurs in a subterm of $A$, the induction hypothesis applies immediately since the translation of $A$ features the translation of every subterm of $A$.

If the reduction occurs at the top-level of $A$, we distinguish three cases:

**$\delta$-reduction**: the reduction $[\![ (A \wr B)\, C ]\!] \mapsto_{\beta} [\![ A\, C \wr B\, C ]\!]$ uses exactly one $\beta$-reduction: it consists in instantiating by $C$ the first variable of $\overline{x}_{(1..\alpha)}$ in $[\![ A \wr B ]\!]$.

**$\rho$-reduction**: as for the previous lemma, the property we check in fact is that $[\![ (\lambda\overline{P}.A)\,\overline{B} ]\!] \mapsto_{\rho} [\![ A\overline{\theta_{(P \ll B)}} ]\!]$ for any set of patterns $\overline{P}$. We proceed by induction over $\overline{P}$ using the order induced by $\#$.

**if** $P \equiv x$ we have $[\![ (\lambda x.A)B ]\!] = (\lambda x.[\![ A ]\!])[\![ B ]\!] \mapsto_{\beta} [\![ A ]\!][x := [\![ B ]\!]]$ and by Lemma 4 this term equals $[\![ A[x := B] ]\!]$.

**if** $P \equiv f\, P_1\, \dots\, P_p$ then

$$\begin{aligned} (\lambda(f\, P_1\, \dots\, P_p).A)\,(f\, B_1\, \dots\, B_p) &\mapsto_{\rho} A\theta_{(f\, P_1\, \dots\, P_p \ll f\, B_1\, \dots\, B_p)} \\ &= A\theta_{(P_1 \ll B_1)} \dots \theta_{(P_p \ll B_p)} \end{aligned}$$

The simulation of this reduction begins with the following $\beta$-reductions, where $\alpha_f$ and $p$ may be zero, but at least the last $\beta$-reduction (instantiating $z$) takes place.

$$\begin{array}{lll} & [\![ \, (\lambda(f\,\overline{P}_{(1..p)}).A)\,(f\,\overline{B}_{(1..p)}) \, ]\!] & \\ = & \Big(\big(\lambda\overline{x}_{(1..\alpha_f)}.\ \lambda z.(z\overline{x}_{(1..x_{\alpha_f})})\big)\overline{[\![ B ]\!]}_{(1..p)}\Big)\ \overline{x_\perp}_{(p+1..\alpha_f)}\ [\![ \lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A ]\!] & \\ \mapsto\!\!\!\!\rightarrow_\beta & \Big(\lambda\overline{x}_{(p+1..\alpha_f)}.\ \lambda z.(z\overline{[\![ B ]\!]}_{(1..p)}\overline{x}_{(p+1..\alpha_f)})\Big)\ \overline{x_\perp}_{(p+1..\alpha_f)}\ [\![ \lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A ]\!] & \\ \mapsto\!\!\!\!\rightarrow_\beta & (\lambda z.(z\overline{[\![ B ]\!]}_{(1..p)}\overline{x_\perp}_{(p+1..\alpha_f)}))[\![ \lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A ]\!] & \\ \mapsto\!\!\!\!\rightarrow_\beta & [\![ \lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A ]\!]\overline{[\![ B ]\!]}_{(1..p)}\overline{x_\perp}_{(p+1..\alpha_f)} & \end{array}$$

Notice that, when considering non-linear patterns, if matching is successful, then every subterm of the argument corresponding to a same variable are equal. That justifies that, in the translation, we can choose any of these subterms, and the other ones can instantiate fresh variables not occurring in the body $A$ of the abstraction.

By induction hypothesis over $P_1 \dots P_p$, the $p$ reductions corresponding to those new patterns are correctly simulated. The variables $x'_{p+1} \dots x'_{\alpha_f}$ are instantiated by $x_\perp$, but since they do not appear into $[\![ A ]\!]$ they do not affect the further reductions of $[\![ A ]\!]$. We have then

$$\begin{array}{lll} & [\![ \, (\lambda(f\,\overline{P}_{(1..p)}).A)\,(f\,\overline{B}_{(1..p)}) \, ]\!] & \\ \mapsto\!\!\!\!\rightarrow_\beta & [\![ \lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A ]\!]\ \overline{[\![ B ]\!]}_{(1..p)}\overline{x_\perp}_{(p+1..\alpha_f)} & \\ = & [\![ \, (\lambda\overline{P}_{(1..p)}.\lambda\overline{x'}_{(p+1..\alpha_f)}.A)\ \overline{B}_{(1..p)} \, ]\!]\overline{x_\perp}_{(p+1..\alpha_f)} & \\ \mapsto\!\!\!\!\rightarrow_\beta & [\![ A\theta_{(P_1 \ll B_1)} \dots \theta_{(P_p \ll B_p)} ]\!] & \text{(by induction hypothesis)} \\ = & [\![ A\theta_{(f\ P_1\ \dots\ P_p \ll f\ B_1\ \dots\ B_p)} ]\!] & \end{array}$$

□

Now that we have shown how to translate terms preserving reductions, let us see how types can be dealt with. First we discuss the main issues that appear, and then we give the full typed translation.

## 6. The typed translation

To begin this section, we explain on some examples three key issues that appear in the typed translation:

1 how to type (in $\mathsf{F}\omega$) the translation of a $P^2TS$ constant (accounting for the use of terms depending on types in the target language);
2 how to type (in $\mathsf{F}\omega$) the translation of a $P^2TS$ variable (accounting for the use of types depending on types in the target language);
3 how to translate the matching constraints appearing in the $\rho$-types (see Def. 4 for details on what is a $\rho$-type).

We will see that the general structure of the previous untyped translation can be preserved, at the cost of adding some suitable type abstractions and instantiations. Then we will detail a full typed translation and prove that it is correct with respect to reductions and typing.

**Notations** For the rest of the paper, we adopt the following abbreviation, for any types $\sigma, \sigma_1, \ldots, \sigma_\alpha$ in $\mathsf{F}\omega$.

$$\bigwedge \overline{\sigma}_{(1..\alpha)} \triangleq \Pi\beta{:}*.\big((\overline{\sigma}_{(1..\alpha)} \to \beta) \to \beta\big)$$

In particular, when $\alpha = 0$:

$$\bigwedge \emptyset \triangleq \Pi\beta{:}*.(\beta \to \beta)$$

and when $\alpha = 1$:

$$\bigwedge \sigma \triangleq \Pi\beta{:}*.((\sigma \to \beta) \to \beta)$$

6.1. *Typing the translation of a constant*

Let us explain why the previous translation of constants cannot be typed without terms depending on types; first we study a simple example, then we will generalize to any constant.

Suppose that $f$ is a constant with type $\Pi x{:}\iota.\iota$ in $\rho_\to$, that $B$ is a $\rho$-term such that $\vdash_\Sigma B : \iota$ (in $\rho_\to$), and finally that $A$ is a $\rho$-term typable in $\rho_\to$.

Clearly, $(\lambda(fx).A)(fB)$ has the same type as $A$. If typing is preserved by the translation, we should have $\vdash_{\mathsf{F}\omega} [\![ A ]\!] : \tau$ and $\vdash_{\mathsf{F}\omega} [\![ (\lambda(fx).A)(fB) ]\!] : \tau$ for a certain $\tau$ in $\mathsf{F}\omega$.

Let us determine how $[\![ (\lambda(fx).A)(fB) ]\!] = [\![ (\lambda(fx).A) ]\!][\![ (fB) ]\!]$ is typed in $\mathsf{F}\omega$.

We have $\alpha_f = 1$, so the translation of $f$ is $\lambda x_1.\lambda z.(z\ x_1)$ and its type is shown below.

$$\vdash_{\mathsf{F}\omega} \lambda x_1.\lambda z.(z\ x_1) : \sigma \to (\sigma \to \beta) \to \beta$$

Here $\sigma$ is supposed to be a correct type for the argument of $[\![ f ]\!]$ and $\beta$ is some unknown type to be precised, when we find more information on the type of $z$.

If typing is preserved by the translation, we should have $\vdash_{\mathsf{F}\omega} [\![ B ]\!] : \sigma$ (since we defined $\sigma$ as the type expected by $[\![ f ]\!]$), and thus $\vdash_{\mathsf{F}\omega} [\![ f\, B ]\!] : (\sigma \to \beta) \to \beta$ .

To determine the value of $\beta$, let us study similarly the type of the translated abstraction $[\![ \lambda(f\, x).A ]\!] = \lambda u.\,\big(u\big(\lambda x.[\![ A ]\!]\big)\big)$:

$$\vdash_{\mathsf{F}\omega} \lambda u.\,\big(u\big(\lambda x.[\![ A ]\!]\big)\big) : ((\sigma \to \tau) \to \gamma) \to \gamma$$

where we have defined $\tau$ as the type of $[\![ A ]\!]$, and $\gamma$ is some unknown type to be precised, when we find more information on the type of $u$. We see that $[\![ \lambda(f\, x).A ]\!]$ can be applied to $[\![ f\, B ]\!]$ if and only if

$$(\sigma \to \tau) \to \gamma \;=\; (\sigma \to \beta) \to \beta$$

which reduces to the simple condition $\tau = \beta = \gamma$. However $\beta$ appears in the type of $[\![ f ]\!]$ and $\tau$ in the type of $[\![ A ]\!]$. Since it is impossible to guess what function will be applied to a given term, we have to make this translation more versatile: this leads us to introducing the polymorphism of Girard's System $\mathsf{F}$ in the target language.

By taking $\beta$ as a type variable, we can write (with the type abbreviations defined above):

$$\begin{aligned} [\![ f ]\!] &\triangleq \lambda x_1.\lambda\beta{:}*.\lambda z.(z x_1) : \sigma \to \bigwedge \sigma \\ [\![ \lambda(f\, x).A ]\!] &\triangleq \lambda u.(u\ \tau\ \lambda x.[\![ A ]\!]) : (\bigwedge \sigma) \to \tau \end{aligned}$$

and then $\vdash_{\mathsf{F}\omega} [\![ \lambda(f\,x).A ]\!][\![ f\,B ]\!] : \tau$ for any $B$ such that $[\![ B ]\!] : \sigma$.

The general case is really similar: every term $[\![ f ]\!]$ has a type looking like

$$\sigma_1 \to \dots \sigma_{\alpha_f} \to \bigwedge \overline{\sigma}_{(1..\alpha_f)}$$

Another interest of polymorphism is that the variable $x_\perp$ can be used whenever we want a placeholder with an arbitrary type: we will take the type of $x_\perp$ to be $\Pi(\iota : *).\iota$ (also denoted $\perp$), so that if $[\![ \Gamma ]\!] \vdash_{\mathsf{F}\omega} \sigma : *$, then $[\![ \Gamma ]\!] \vdash_{\mathsf{F}\omega} x_\perp \sigma : \sigma$. With this simple trick, the use of $x_\perp$ we had made in the untyped translation becomes compatible with typing. To conform with the typing assumptions made when typing an abstraction, the various placeholders $x_\perp$ are given the type of $x'_n$ in order to fit with the type expected by $u$.

### 6.2. *Typing the translation of a variable*

In this subsection, we explain that we will need a new type variable $\beta_x$ for each variable $x$ appearing in a $\rho$-term (including bound variables appearing in a type). The main reason is that one can not predict which term $A$ will instantiate a variable $x$, and even if some terms $A_1, A_2, \dots$ have the same type in $\rho_\to$, their translation can have many different types. Consider the following examples (with $\Sigma \equiv \iota{:}*, a{:}\iota, f : \Pi(y{:}\iota).\iota$):

$$\begin{aligned} x : \Pi y{:}\iota.\iota \;&\vdash_\Sigma x \quad\;\; : \Pi y{:}\iota.\iota \\ &\vdash_\Sigma \lambda y{:}\iota.y : \Pi y{:}\iota.\iota \\ &\vdash_\Sigma \lambda y{:}\iota.a : \Pi y{:}\iota.\iota \\ &\vdash_\Sigma f \quad\;\; : \Pi y{:}\iota.\iota \end{aligned}$$

The three terms $\lambda y.y$, $\lambda y.a$ and $f$ can instantiate $x$ since they have the same type. However, supposing the type $\iota$ is translated by a type $\beta_y$, the translation gives:

$$\begin{aligned} &\vdash_{\mathsf{F}\omega} \lambda y{:}\beta_y.y \quad\;\; : \beta_y \to \beta_y \\ &\vdash_{\mathsf{F}\omega} \lambda y{:}\beta_y.[\![ a ]\!] : \beta_y \to \textstyle\bigwedge \emptyset \\ &\vdash_{\mathsf{F}\omega} [\![ f ]\!] \quad\;\; : \beta_y \to \textstyle\bigwedge \beta_y \end{aligned}$$

The shape of the type we obtain is always the same, but the return type changes; it may not even use $\beta_y$. Thus, instead of having $\vdash_{\mathsf{F}\omega} x : \beta_y \to \beta_y$, we add a type variable $\beta_x$ which allows us to build the return type as needed. The translation of $\Pi y{:}\iota.\iota$ then becomes $\beta_y \to \beta_x\beta_y$, where $\beta_x$ can be instantiated with a type depending on a type, which justifies the use of $\mathsf{F}\omega$.

Henceforth, when translating an abstraction $\lambda x.A$, we shall add the variable $\beta_x$ to the context; and when translating an application $A\,B$, we shall abstract on $\beta_x$ and give an additional argument in order to build the proper return type.

In our examples, supposing that $\beta_y : *$, the variable $\beta_x$ must be instantiated as follows:

| | | |
|---|---|---|
| $\lambda y.y$ | $\beta_x := \lambda\beta{:}* .\beta$ | $(\lambda\beta{:}* .\beta)\,\beta_y \mapsto_\beta \beta_y$ |
| $\lambda y.a$ | $\beta_x := \lambda\beta{:}* .\bigwedge\emptyset$ | $(\lambda\beta{:}* .\bigwedge\emptyset)\,\beta_y \mapsto_\beta \bigwedge\emptyset$ |
| $f$ | $\beta_x := \lambda\beta{:}* .\bigwedge\beta$ | $(\lambda\beta{:}* .\bigwedge\beta)\,\beta_y \mapsto_\beta \bigwedge\beta_y$ |

We will design a function $\mathbb{K}(\cdot\ ;\ \cdot)$ to compute a suitable kind for $\beta_x$ according to the kinds of the type variables $\beta_y$ which are the arguments of $\beta_x$. For instance, here, we should take $\beta_x : * \to *$.

### 6.3. *Translating matching constraints appearing in $\rho$-types*

A third issue with the typed translation is the presence of matching constraints in $\rho_\to$ types. When such a constraint can be resolved, we will simplify it systematically to be as close as possible to the case of types with the shape of a Π-abstraction.

However, if a term $(\lambda P{:}\Delta.A)\,B$ has type $[P \ll_\Delta B]C$ and this constraint can not be resolved, we will represent it as follows: the term will be translated into $[\![\,\lambda P{:}\Delta.A\,]\!](w[\![\,B\,]\!])$, where $w$ is a fresh variable called on purpose *postponement variable*. Taking $\vdash_{\mathsf{F}\omega} w : \sigma \to \tau_1$ if $\vdash_{\mathsf{F}\omega} [\![\,B\,]\!] : \sigma$, we will have indeed $\vdash_{\mathsf{F}\omega} [\![\,(\lambda P{:}\Delta.A)\,B\,]\!] : \tau_2$ but the reduction will be frozen until $w$ is instantiated.

If further reductions transform $B$ into $B'$ such that the constraint can be resolved, that will ensure that $P$ and $B'$ have a same type; then we will instantiate $w$ with the identity, which will erase the encoding of the constraint in the translation too. This mechanism will be detailed in the typed translation.

### 6.4. *The full typed translation*

As we will produce typed $\lambda$-terms, we will index every $\lambda$-abstraction with the type of the corresponding bound variable. Thus, the first step of the typed translation is to define a translation for some types.

Recall that, according to Lemma 2, in the $P^2TS$ system we consider, types can be well identified, which allows us to stratify the translations throughout the paper.

As explained before, for each variable $x$ in the $\rho$-term we translate, we will use a type variable $\beta_x$. The function $\mathbb{K}(C\ ;\ \overline{k})$ computes a correct kind for this new variable, using $\overline{k}$ as an accumulator for the kinds of the variables appearing in $C$.

If the original variable $x$ has type $C$, the type of $\beta_x$ will be found by computing $\mathbb{K}(C\ ;\ )$ (*i.e.* starting with an empty accumulator).

**Definition 6 (Kind of the type variable associated to a $P^2TS$ variable).**

$$\begin{aligned}
\mathbb{K}(\iota \ ; \ \overline{k}) &\triangleq \overline{k} \to * \\
\mathbb{K}(\Pi P{:}\Delta.C \ ; \ \overline{k}) &\triangleq \mathbb{K}(C \ ; \ \overline{k}, \overline{\mathbb{K}(C_y \ ; \ )}_{((y:C_y)\in\Delta)}) \\
\mathbb{K}([P \ll_\Delta B]C \ ; \ \overline{k}) &\triangleq \mathbb{K}(C \ ; \ \overline{k})
\end{aligned}$$

Then, we can define the translations of types. We need three mutually dependent definitions : $(\!|P|\!)$ gives the type of the variable $u$ that appears in $[\![ \lambda P.A ]\!]$ ; we use $[\![ C ]\!]^x_{\overline{\gamma}}$ to translate the type $C$ of a variable $x$ ; similarly $[\![ C ]\!]^f_{\overline{\tau}}$ translates the type $C$ of a constant $f$.

First $(\!|P \, ; \, \Delta|\!)$ finds the type of the variable $u$ that appears in $[\![ \lambda P{:}\Delta.A ]\!]$.

**Definition 7 (Type of the variable associated to a pattern).**

$$(\!|f \, \overline{P} \, ; \, \Delta|\!) \triangleq [\![ C ]\!]^f_{\overline{(\!|P \, ; \, \Delta|\!)}} \qquad \text{if } \Delta \vdash_\Sigma f \, \overline{P} : C$$

$$(\!|x \, ; \, \Delta|\!) \triangleq [\![ C ]\!]^x \qquad \text{if } \Delta \vdash x : C$$

To lighten notations, we will generally forget $\Delta$ and write $(\!|P|\!)$.

For instance we will have

$$(\!|f \, (g \, x_1) \, x_2|\!) = \bigwedge \left( (\wedge [\![ \sigma_1 ]\!]^{x_1}), [\![ \sigma_2 ]\!]^{x_2} \right)$$

Again, as for the encoding of matching, the shape of patterns is preserved but the head constants are forgotten.

Then, $[\![ C ]\!]^x_{\overline{\gamma}}$ translates the type $C$ supposing it is the type of a variable $x$. The various new type variables and postponement variables will be added to the context.

**Definition 8 (Translation for the type of a $P^2TS$ variable).**

$$\begin{aligned}
[\![ \iota ]\!]^x_{\overline{\gamma}} &\triangleq \beta_x \overline{\gamma} && \text{if } \iota \text{ is atomic} \\
[\![ \Pi P{:}\Delta.C ]\!]^x_{\overline{\gamma}} &\triangleq (\!|P|\!) \to [\![ C ]\!]^x_{\overline{\gamma}, \overline{\beta_y}} && \text{with the } \overline{\beta_y} \text{ chosen so that } y \text{ ranges over } \mathcal{D}om(\Delta)
\end{aligned}$$

$$[\![ [P \ll_\Delta B]C ]\!]^x_{\overline{\gamma}} \triangleq [\![ C ]\!]^x_{\overline{\gamma}}$$

Finally $[\![ C ]\!]^f_{\overline{\tau}}$ translates the type $C$ when it is the type of a constant, and it differs from $[\![ C ]\!]^x_{\overline{\gamma}}$ only by the return type, which will be some $\bigwedge \overline{\tau}$ instead of $\beta_x \overline{\gamma}$.

**Definition 9 (Translation for the type of a $P^2TS$ constant).**

$$\begin{aligned}
[\![ \iota ]\!]^f_{\overline{\tau}} &\triangleq \bigwedge \overline{\tau} && \text{if } \iota \text{ is atomic} \\
[\![ \Pi P{:}\Delta.C ]\!]^f_{\overline{\tau}} &\triangleq (\!|P|\!) \to [\![ C ]\!]^f_{\overline{\tau}, \, (\!|P|\!)} \\
[\![ [P \ll_\Delta B]C ]\!]^f_{\overline{\tau}} &\triangleq [\![ C ]\!]^f_{\overline{\tau}}
\end{aligned}$$

We can now give the translation of contexts for which the reader is reported to Fig. 5.

The translation of $\Gamma$ knowing we are typing $[\![ A ]\!]$ is denoted $[\![\Gamma/A; C]\!]$, where we will need some additional variables. The third argument $C$ is the type of $A$, which in some cases needs to be browsed to translate the context. The base case is $[\![\Gamma/\ ;\ ]\!]$ which is common to every translation. As said above, we use $x_\perp$ with type $\perp$.

$$
\begin{array}{rcl}
[\![\emptyset/\ ;\ ]\!] & \triangleq & x_\perp : \perp \\
[\![\Gamma, x{:}C/\ ;\ ]\!] & \triangleq & [\![\Gamma/\ ;\ ]\!],\ \beta_x : \mathbb{K}(C\ ;\ ),\ x : [\![ C ]\!]^x \qquad (\text{ if } \Gamma \vdash_\Sigma C : *) \\
[\![\Gamma, x{:}*/\ ;\ ]\!] & \triangleq & [\![\Gamma/\ ;\ ]\!],\ x : *
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma/x; \Pi P_n{:}\Delta_n.C]\!] & \triangleq & [\![\Gamma/x; C]\!], \overline{\beta_y{:}\mathbb{K}(C_y\ ;\ )}_{((y:C_y)\in\Delta_n)} \\
[\![\Gamma/x; [P \ll_\Delta B]C]\!] & \triangleq & [\![\Gamma/x; C]\!] \\
[\![\Gamma/x; \iota]\!] & \triangleq & [\![\Gamma/\ ;\ ]\!], \beta_x{:}\mathbb{K}(C_x\ ;\ )
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma/f; \Pi P_n{:}\Delta_n.C]\!] & \triangleq & [\![\Gamma/f; C]\!], \overline{\beta_y{:}\mathbb{K}(C_y\ ;\ )}_{((y:C_y)\in\Delta_n)} \\
[\![\Gamma/f; [P \ll_\Delta B]C]\!] & \triangleq & [\![\Gamma/f; C]\!] \\
[\![\Gamma/f; \iota]\!] & \triangleq & [\![\Gamma/\ ;\ ]\!]
\end{array}
$$

$$
[\![\Gamma/A \wr B; C]\!] \triangleq [\![\Gamma/A; C]\!], [\![\Gamma/B; C]\!]
$$

$$
[\![\Gamma/\lambda x{:}(x{:}C).A; \Pi x{:}C.D]\!] \triangleq [\![\Gamma, x{:}C/A; D]\!] \setminus x
$$

$$
[\![\Gamma/\lambda P{:}\Delta.A; \Pi P{:}\Delta.C]\!] \triangleq [\![\Gamma, \Delta/A; C]\!] \setminus \mathcal{D}om(P)
$$

$$
\begin{array}{rcl}
[\![\Gamma/A\,B; [P \ll_\Delta B]C]\!] & \triangleq & \left([\![\Gamma/A; \Pi P{:}\Delta.C]\!] \setminus \overline{\beta_x}, \overline{w}\right)\ ,\ [\![\Gamma/B; D]\!]\ ,\ \overline{w'} \\
& & \text{if there exists } \overline{\tau_x},\ [\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![ B ]\!] : (\!|P|\!)[\overline{\beta_x := \tau_x}]_{x\in\mathcal{D}om(\Delta)} \\
& & \text{where } \Gamma \vdash_\Sigma B : D \text{ as in the type derivation for } [P \ll_\Delta B]C \\
& & \text{where } \overline{\beta_x}, \overline{w} \text{ are the variables updated by } P \ll_\Delta B \\
& & \text{and } \overline{w'} \text{ are the postponement variables created by the update}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\Gamma/A\,B; [P \ll_\Delta B]C]\!] & \triangleq & [\![\Gamma/A; \Pi P{:}\Delta.C]\!]\ ,\ [\![\Gamma/B; D]\!]\ ,\ w_{(P\ll B)} : \sigma \to (\!|P|\!) \\
& & \text{if } [\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![ B ]\!] : \sigma \\
& & \text{and there exists no } \overline{\tau_x},\ \sigma =_\beta (\!|P|\!)[\overline{\beta_x := \tau_x}]_{x\in\mathcal{D}om(\Delta)} \\
& & \text{where } \Gamma \vdash_\Sigma B : D \text{ as in the type derivation for } [P \ll_\Delta B]C)
\end{array}
$$

Fig. 5. Translation of contexts.

The translation of terms is given in Fig. 6.

$$
\begin{aligned}
[\![\, x \,]\!] &\triangleq x \\
[\![\, f \,]\!] &\triangleq \lambda\overline{x{:}\,(\!|P|\!)}_{(1..\alpha_f)}.\lambda\beta{:}\,*\,.\lambda z{:}(\overline{(\!|P|\!)}_{(1..\alpha_f)} \to \beta).(z\overline{x}_{(1..\alpha_f)}) \\
&\quad \text{where } \vdash_\Sigma f : \Pi\overline{P{:}\Delta}_{(1..\alpha_f)}.\iota \\
[\![\, A \wr B \,]\!] &\triangleq \lambda\overline{x{:}\,(\!|P|\!)}_{(1..\alpha_{A\wr B})}.\left(\lambda z{:}\sigma.[\![\, A \,]\!]\,\overline{x}_{(1..\alpha_{A\wr B})}\right)\,([\![\, B \,]\!]\,\overline{x}_{(1..\alpha_{A\wr B})}) \\
&\quad \text{where } \Gamma \vdash_\Sigma A \wr B : \Pi\overline{P{:}\Delta}_{(1..\alpha_{A\wr B})}.\iota \\
&\quad \text{and } [\![\Gamma/B; \Pi\overline{P{:}\Delta}_{(1..\alpha_{A\wr B})}.\iota]\!] \vdash_{\mathsf{F}\omega} [\![\, B \,]\!]\overline{x} : \sigma \\
[\![\, \lambda x{:}(x{:}C).A \,]\!] &\triangleq \lambda x{:}[\![\, C \,]\!]^x\,.\,(\lambda\overline{y{:}\tau}.[\![\, A \,]\!])\,\overline{[\![\, D \,]\!]} \\
&\quad \text{where } \overline{D} \text{ are the terms appearing in } C \text{ and } \overline{[\![\,\Gamma\,]\!] \vdash_{\mathsf{F}\omega} [\![\, D \,]\!] : \tau} \\
[\![\, \lambda(f\,\overline{P}_{(1..p)}){:}\Delta.A \,]\!] &\triangleq \lambda u{:}(\!|f\,\overline{P}|\!).(u\overline{(x_\perp\,(\!|P'|\!))}_{(p+1..\alpha_f)}\ \ \tau\ [\![\,\overline{\lambda P{:}\Delta}_{(1..p)}\overline{\lambda x'{:}(x{:}(\!|P'|\!))}_{(p+1..\alpha_f)}.A\,]\!]) \\
&\quad \text{where } \Delta \vdash_\Sigma f\,\overline{P}_{(1..p)} : \Pi\overline{P'{:}\Delta}_{(p+1..\alpha_f)}.\iota \quad \text{and} \quad [\![\Gamma/A; C]\!] \vdash_{\mathsf{F}\omega} [\![\, A \,]\!] : \tau \\
&\quad \text{where } \Gamma \vdash_\Sigma A : C \text{ as in the type derivation for } \lambda(f\,\overline{P}_{(1..p)}){:}\Delta.A \\
[\![\, A\,B \,]\!] &\triangleq \left(\lambda\overline{\beta_x}.\lambda\overline{w}.[\![\, A \,]\!]\right)\,\overline{\theta(\beta_x)}\,\overline{\theta(w)}\,[\![\, B \,]\!] \\
&\quad \text{where } \overline{\beta_x}, \overline{w} \text{ are the variables updated by } P \ll B \\
&\qquad \text{if there exists } \overline{\tau_x},\ [\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![\, B \,]\!] :\ (\!|P|\!)[\overline{\beta_x := \tau_x}]_{x\in\mathcal{D}om(\Delta)} \\
[\![\, A\,B \,]\!] &\triangleq [\![\, A \,]\!]\,(w_{(P\ll B)}\,[\![\, B \,]\!]) \\
&\qquad \text{if there exists no } \overline{\tau_x},\ [\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![\, B \,]\!] :\ (\!|P|\!)[\overline{\beta_x := \tau_x}]_{x\in\mathcal{D}om(\Delta)}
\end{aligned}
$$

Fig. 6. Typed translation.

In the translation of the abstraction over a variable $x$, we add some redexes accounting for the terms appearing in the type of $x$, so that the reductions occurring in a context $\Delta$ are not forgotten in the translation (in a given type the number of $\sigma$-reductions is bounded but in $[P \ll B]C$ an infinite reduction of $B$ can occur). This technique is common when translating systems where the types can feature terms.

The translation of application in the case of a solvable constraint uses a notion of *variable update* generated by a constraint, defined as follows.

**Definition 10 (Variable update generated by a constraint).**
Let $P \ll_\Delta B$ be a matching constraint.
The *variable update* generated by this constraint is a substitution $\theta$ such that:

— If $[\![\Gamma/B; ]\!] \vdash_{\mathsf{F}\omega} [\![\, B \,]\!] :\ (\!|P|\!)[\overline{\beta_x := \tau_{x(x\in\mathcal{D}om(\Delta))}}]$, then

1 For every $x \in \mathcal{D}om(\Delta)$, we have $\theta(\beta_x) = \tau_x$.

2 Let $\theta(w_{(P\ll B)}) = \lambda x{:}\,(\!|P|\!)[\overline{\beta_x := \tau_{x(x\in\mathcal{D}om(\Delta))}}].x$

3 For every $w_{Q\ll D}$ whose type features a variable $\beta_x \in \mathcal{D}om(\theta)$, add to $\theta$ the variable update generated by $Q \ll D\theta$.

— Otherwise, $\theta(w_{(P\ll B)}) = w'_{(P\ll B)}$ with type $\sigma \to (\!|P|\!)\theta$, where $\sigma$ is the type of $[\![\, B \,]\!]$.

Notice $\theta$ is built recursively, and this recursion is well-founded: the constraint $P \ll B$ can update a variable $w_{Q \ll D}$ only if $\mathcal{FV}(D) \cap \mathcal{FV}(P) \neq \emptyset$, which generates a well-founded order over the constraints. Notice too that:

— $\theta$ associates a type to every type variable;
— $\theta$ associates either a new postponement variable or the identity to every postponement variable.

## 7. Correctness of the typed translation

Finally, we can prove that our translation preserves reductions and typability. These technical lemmas flesh out the sketch of proof we gave for Theorem 2.

**Lemma 5 (Well-kindedness).**
If $\Gamma \vdash_\Sigma C : *$ then $[\![\Gamma/x; C]\!] \vdash_{\mathsf{F}\omega} [\![ C ]\!]^x : *$ for some fresh variable $x$.

*Proof.* Nearly immediate. The definition of $\mathbb{K}(C \ ; \ \overline{k})$ ensures that each $\beta_x$ has the suitable kind. □

**Theorem 5 (Typed faithful translation).**
For all $\Sigma, \Gamma, A, C$, if $\Gamma \vdash_\Sigma A : C : *$ then, for some fresh variable $z$,
there exists $\tau_A$, $[\![\Gamma/A; C]\!] \vdash_{\mathsf{F}\omega} [\![ A ]\!] : [\![ C ]\!]^z[\beta_z := \tau_A]$

*Proof.* We prove by case analysis on term $A$ that $\tau_A$ can be defined as follows:

$$
\begin{array}{rcll}
\tau_x & \triangleq & \beta_x & \\
\tau_f & \triangleq & \lambda\overline{\beta_y}.\bigwedge \overline{(\!|P_n|\!)}_{(1..\alpha_f)} & \text{if } f : \Pi P_1 \dots \Pi P_{\alpha_f}.\iota \\
\tau_{A \imath B} & \triangleq & \tau_A & \\
\tau_{\lambda P:\Delta.A} & \triangleq & \lambda\overline{\beta_x}.\tau_A & \text{if } \mathcal{FV}(P) = \overline{x} \\
\tau_{A\,B} & \triangleq & \tau_A\,\overline{\tau_y} & \text{if there exists } \overline{\tau_y},\ [\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![ B ]\!] : (\!|P|\!)[\overline{\beta_y := \tau_y}] \\
\tau_{A\,B} & \triangleq & \tau_A\,\overline{\beta_x} & \text{otherwise}
\end{array}
$$

It is easy to see that if two types $C$ and $C'$ are convertible modulo $=_{\beta\delta}$, then $[\![ C ]\!]^z$ and $[\![ C' ]\!]^z$ are convertible modulo $=_\beta$, so we can choose any representative of the type of $A$. The proof proceeds by induction on the structure of term $A$ which does not contain any matching constraint.

**For a variable $x$ we have**

$$(\textsc{Var})\ \frac{\Gamma \vdash_\Sigma C : s}{\Gamma, x{:}C \vdash_\Sigma x : C}$$

The sort $s$ must be $*$. By Lemma 5, we have $[\![\Gamma/x; C]\!] \vdash_{\mathsf{F}\omega} [\![ C ]\!]^x : *$, hence

$$(\textsc{Var})\ \frac{[\![\Gamma/\ ;\ ]\!], \beta_x : \mathbb{K}(C \ ; \ ) \vdash_{\mathsf{F}\omega} [\![ C ]\!]^x : *}{[\![\Gamma, x{:}C/\ ;\ ]\!] \vdash_{\mathsf{F}\omega} x : [\![ C ]\!]^x}$$

**For a constant $f$ we have**

$$(\textsc{Const})\ \frac{\Sigma\ sig \qquad f : C \in \Sigma}{\vdash_\Sigma f : C}$$

Then in the derivation of $\Sigma\ sig$, we can find $\vdash_{\Sigma'} C : s$ for some prefix $\Sigma'$ of $\Sigma$, and the sort $s$ must be $*$. We must show that $[\![\emptyset/f;C]\!] \vdash_{\mathsf{F}\omega} [\![\, f \,]\!] : [\![\, C \,]\!]^f$. The matching constraints appearing in $C$ have not been translated in $[\![\, C \,]\!]^f$, so it is sufficient to notice that the variables $\overline{x}_{(1..\alpha_f)}$ bound in $[\![\, f \,]\!]$ have the expected corresponding types $\overline{(\!|P|\!)}_{(1..\alpha_f)}$.

Finally, by definition of $[\![\, C \,]\!]^f$, its return type is $\bigwedge \overline{(\!|P|\!)}_{(1..\alpha_f)}$, which is indeed a valid type for $\lambda\beta{:}*.\lambda z.(z\overline{x}_{(1..\alpha_f)})$.

Let us express this type as $[\![\, C \,]\!]^z[\beta_z := \tau_f]$ for some $\tau_f$. It is sufficient that $\tau_f$ builds the $(\!|P|\!)$ using the $\overline{\beta_y}$ occurring into $[\![\Gamma/f;C]\!]$, which is immediate (modulo $\alpha$-conversion of the $\beta_y$) :

$$\tau_f \triangleq \lambda\overline{\beta_y}.\bigwedge \overline{(\!|P_n|\!)}_{(1..\alpha_f)}$$

Notice that, by Lemma 5 and by definition of $\bigwedge$, we have $[\![\, \Delta \,]\!] \vdash_{\mathsf{F}\omega} \bigwedge \overline{(\!|P_n|\!)}_{(1..\alpha_f)} : *$ so $\tau_f$ has indeed the same kind as $\beta_z$.

**For a structure $A \wr B$ we have**

$$(\textsc{Struct})\ \frac{\Gamma \vdash_\Sigma A : C \qquad \Gamma \vdash_\Sigma B : C}{\Gamma \vdash_\Sigma A \wr B : C}$$

By induction hypothesis $[\![\Gamma/A;C]\!] \vdash [\![\, A \,]\!] : [\![\, C \,]\!]^z[\beta_z := \tau_A]$ and $[\![\Gamma/B;C]\!] \vdash [\![\, B \,]\!] : [\![\, C \,]\!]^z[\beta_z := \tau_B]$. Recall that the structure is not translated as the usual pair of the $\lambda$-calculus but as $(\lambda z.[\![\, A \,]\!])[\![\, B \,]\!]$, hence the type of $[\![\, A \wr B \,]\!]$ is also the type of $[\![\, A \,]\!]$, hence $\tau_{A\wr B} = \tau_A$.

We still have to see that the variables $x_1 \ldots x_{\alpha_{A\wr B}}$ bound in $[\![\, A \wr B \,]\!]$ have respectively types $(\!|P_n|\!)$, which is immediate. The context $[\![\Gamma/A \wr B;C]\!]$ is defined as the union of two contexts, which allows to type the whole term $[\![\, A \wr B \,]\!]$, provided the same $\beta_y$ are used into $[\![\, A \,]\!]$, $[\![\, B \,]\!]$ and the types of $\overline{x}_{(1..\alpha_{A\wr B})}$.

**For an abstraction $\lambda P{:}\Delta.A$ we have**

$$(\textsc{Abs})\ \frac{\Gamma, \Delta \vdash_\Sigma A : C \qquad \Gamma \vdash_\Sigma \Pi P{:}\Delta.C : s}{\Gamma \vdash_\Sigma \lambda P{:}\Delta.A : \Pi P{:}\Delta.C}$$

Then necessarily $s \equiv *$. We discuss according to pattern $P$.

If $P$ is only a variable $x$ then in the translation $x$ has type $[\![\, C_x \,]\!]^x$. By induction hypothesis $[\![\Gamma, x{:}C_x/A;C]\!] \vdash_{\mathsf{F}\omega} [\![\, A \,]\!] : [\![\, C \,]\!]^z[\beta_z := \tau_A]$, and the terms $[\![\, D \,]\!]$ are typable (it is the only information needed to type the redexes $(\lambda y.\ldots)[\![\, D \,]\!]$, which do not influence typing elsewhere). We conclude that

$$[\![\Gamma/\lambda x.A;\Pi x.C]\!] \vdash_{\mathsf{F}\omega} [\![\, \lambda x.A \,]\!] : [\![\, \Pi x.C \,]\!]^z[\beta_z := \lambda\beta_x.\tau_A]$$

Otherwise, in $[\![\, \lambda(f\,\overline{P}_{(1..p)}).A \,]\!]$, the variable $u$ has type $(\!|f\,\overline{P}_{(1..p)}|\!)$. It is then sufficient to check that $u\overline{(x_\perp\ (\!|P'|\!))}_{(p+1..\alpha_f)}\ \ \tau\ [\![\, \lambda\overline{P{:}\Delta}_{(1..p)}.\lambda\overline{x'{:}(\!|P'|\!)}_{(p+1..\alpha_f)}.A \,]\!]$ has type $[\![\, C \,]\!]^z[\beta_z := \tau]$ for some $\tau$.

By definition $(\!|f\,\overline{P}_{(1..p)}|\!) \equiv \overline{(\!|P'|\!)}_{(p+1..\alpha_f)} \to \bigwedge \overline{(\!|P|\!)}_{(1..p)}, \overline{(\!|P'|\!)}_{(p+1..\alpha_f)}$.

The arguments $x_\perp\ (\!|P'|\!)$ absorb the first arguments expected by $u$, and then $\tau$ instantiates the type variable bound in $\bigwedge \overline{(\!|P|\!)}_{(1..p)}, \overline{(\!|P'|\!)}_{(p+1..\alpha_f)}$. Finally, by induction

hypothesis, we have

$$[\![\Gamma/\lambda\overline{P}.\lambda\overline{x'}.A;\Pi\overline{P}.\Pi\overline{P'}.C]\!] \vdash_{\mathsf{F}\omega} \\ [\![\,\lambda\overline{P{:}\Delta}_{(1..p)}.\lambda\overline{x'{:}\,(\!|P'|\!)}_{(p+1..\alpha_f)}.A\,]\!] : [\![\,\Pi\overline{P}_{(1..p)}.\Pi\overline{P'}_{(p+1..\alpha_f)}.C\,]\!]^z[\beta_z := \tau]$$

with $\tau \equiv \lambda\overline{\beta_x}.\tau_A$ where $\overline{\beta_x} = \mathcal{FV}(\overline{P},\overline{P'})$. The free variables in $P'$ have been introduced only for the translation, hence we can do without the corresponding $\beta_x$.
We conclude that

$$[\![\Gamma/\lambda f\,\overline{P}_{(1..p)}.A;\Pi f\,\overline{P}_{(1..p)}.C]\!] \vdash_{\Sigma} \\ [\![\,\lambda f\,\overline{P}_{(1..p)}.A\,]\!] : (\!|f\,\overline{P}_{(1..p)}|\!) \to [\![\,C\,]\!]^z[\beta_z := \lambda\overline{\beta_x}_{(x\in\mathcal{FV}(\overline{P}))}.\tau_A]$$

and we have

$$(\!|f\,\overline{P}_{(1..p)}|\!) \to [\![\,C\,]\!]^z[\beta_z := \lambda\overline{\beta_x}_{(x\in\mathcal{FV}(\overline{P}))}.\tau_A] \;=\; [\![\,\Pi f\,\overline{P}_{(1..p)}.C\,]\!]^z[\beta_z := \lambda\overline{\beta_x}_{(x\in\mathcal{FV}(\overline{P}))}.\tau_A]$$

Finally we obtain the result since $\tau_{\lambda P:\Delta.A} \triangleq \lambda\overline{\beta_x}.\tau_A$

**For an application $A\,B$ we have**

$$(\textsc{Appl})\ \frac{\Gamma \vdash_\Sigma A : \Pi P{:}\Delta.C \qquad \Gamma \vdash_\Sigma [P \ll_\Delta B]C : s}{\Gamma \vdash_\Sigma A\,B : [P \ll_\Delta B]C}$$

Then necessarily $s \equiv *$.
We distinguish two cases, according to typed translation of Fig. 6.

**If there exists $\overline{\tau_x}$, $[\![\Gamma/B;D]\!] \vdash_{\mathsf{F}\omega} [\![\,B\,]\!] : (\!|P|\!)[\overline{\beta_x := \tau_x}]$ then**
$[\![\,A\,B\,]\!] = (\lambda\overline{\beta_x}.\lambda\overline{w}.[\![\,A\,]\!])\,\overline{\theta(\beta_x)}\,\overline{\theta(w)}\,[\![\,B\,]\!]$. By induction hypothesis over $A$ we have $[\![\Gamma/A;\Pi P{:}\Delta.C]\!] \vdash_{\mathsf{F}\omega} [\![\,A\,]\!] : [\![\,\Pi P{:}\Delta.C\,]\!]^z[\beta_z := \tau_A]$.
Let us check that it is correct to instantiate some $w_{Q\ll D}$ with the identity: indeed, if the constraint $Q \ll D$ is solvable, then

$$[\![\Gamma/A\,B;[P \ll_\Delta B]C]\!] \vdash_{\mathsf{F}\omega} [\![\,D\,]\!] : (\!|Q|\!)[\overline{\beta_y := \tau_y}_{(y\in\mathcal{FV}(Q))}]$$

and we have added the abstractions $\lambda\overline{\beta_y}$ and the corresponding arguments $\tau_y$. Hence

$$[\![\Gamma/A\,B;[P \ll_\Delta B]C]\!] \vdash_{\mathsf{F}\omega} w_{Q\ll D} : (\!|Q|\!)[\overline{\beta_y := \tau_y}_{(y\in\mathcal{FV}(Q))}] \to (\!|Q|\!)[\overline{\beta_y := \tau_y}_{(y\in\mathcal{FV}(Q))}]$$

and we can instantiate it with $\lambda x{:}\,(\!|Q|\!)[\overline{\beta_y := \tau_y}_{(y\in\mathcal{FV}(Q))}].x$. Then

$$\left([\![\Gamma/A;\Pi P{:}\Delta.C]\!] \setminus \overline{\beta_x},\overline{w}\right),\overline{w'} \vdash_{\mathsf{F}\omega} (\lambda\overline{\beta_x}.\lambda\overline{w}.[\![\,A\,]\!])\,\overline{\tau_x}\,\overline{t} : [\![\,\Pi P{:}\Delta.C\,]\!]^z[\beta_z := \tau_A][\overline{\beta_x := \tau_x}].$$

Now

$$\begin{aligned}
[\![\,\Pi P{:}\Delta.C\,]\!]^z[\beta_z := \tau_A][\overline{\beta_x := \tau_x}] &= (\!|P|\!)[\overline{\beta_x := \tau_x}] \to [\![\,C\,]\!]^z_{\overline{\beta_x}}[\beta_z := \tau_A][\overline{\beta_x := \tau_x}] \\
&= (\!|P|\!)[\overline{\beta_x := \tau_x}] \to [\![\,C\,]\!]^z_{\overline{\tau_x}}[\beta_z := \tau_A] \\
&= (\!|P|\!)[\overline{\beta_x := \tau_x}] \to [\![\,C\,]\!]^z[\beta_z := \tau_A\,\overline{\tau_x}]
\end{aligned}$$

The expected type for the argument corresponds to the type of $[\![\,B\,]\!]$, and the typing context for the application is the union of two contexts, where
$[\![\Gamma/A\,B;[P \ll_\Delta B]C]\!] \vdash_{\mathsf{F}\omega} [\![\,A\,B\,]\!] : [\![\,C\,]\!]^z[\beta_z := \tau_A\,\overline{\tau_x}]$, thus in that case

$$\tau_{A\,B} \triangleq \tau_A\overline{\tau_x}$$

**Otherwise**, $[\![ A\,B ]\!] = [\![ A ]\!](w[\![ B ]\!])$ where $w$ has type $\sigma \to (\!|P|\!)$ (where $\sigma$ is the type of $[\![ B ]\!]$).
By induction hypothesis, $[\![\Gamma/A; \Pi P{:}\Delta.C]\!] \vdash_{\mathsf{F}\omega} [\![ A ]\!] : [\![\Pi P{:}\Delta.C ]\!]^z[\beta_z := \tau_A]$, but

$$\begin{aligned}[\![\Pi P{:}\Delta.C ]\!]^z[\beta_z := \tau_A] &= (\!|P|\!) \to [\![ C ]\!]^z_{\overline{\beta_x}}[\beta_z := \tau_A] \\ &= (\!|P|\!) \to [\![ C ]\!]^z[\beta_z := \tau_A\,\overline{\beta_x}]\end{aligned}$$

The typing context for the application is the union of two contexts in which we have added the variable $w$, hence
$[\![\Gamma/A\,B; [P \ll_\Delta B]C]\!] \vdash_{\mathsf{F}\omega} [\![ A\,B ]\!] : [\![ C ]\!]^z[\beta_z := \tau_A\,\overline{\beta_x}]$, thus

$$\tau_{A\,B} \triangleq \tau_A\overline{\beta_x}$$

□

**Lemma 6 (Typed faithful reductions).**
Lemma 4 (closure by substitution) and Theorem 4 (faithful reductions) are still valid in the typed translation.

*Proof.* The proof resembles closely the untyped case: we have only added $\lambda$-abstractions and applications on types, on terms appearing in the $\rho$-types, and on postponement variables. It is sufficient to check that these ones behave as expected.

1 The only abstractions and applications on types appear in the translation of constants, abstractions and applications. It is immediate to see that every type abstraction has exactly one corresponding type application.
Thus, it is sufficient to check that the type variables have the expected kind (Lemma 5) and that the $\lambda$-terms produced by $[\![\ ]\!]$ are well-typed (Theorem 5).
2 The redexes added in $\lambda x{:}C.A$ allow to translate the reductions of shape $\lambda P{:}\Delta.A \mapsto_{\rho\sigma\delta} \lambda P{:}\Delta'.A$, where $\Delta'$ is $\Delta$ in which a type $C$ has undergone a reduction $\mapsto_{\rho\sigma\delta}$. This type $C$ is necessarily the type of a variable $x$ in $P$, hence in the translation $[\![\lambda P{:}\Delta.A ]\!]$, we translate some $\lambda x{:}C.A'$. Thus, for the subterm $D$ of $C$ in which the reduction occurs, the $\lambda$-term $[\![ D ]\!]$ appears in $[\![\lambda P{:}\Delta.A ]\!]$ hence by induction hypothesis the reduction is also translated into $[\![ D ]\!]$.
When we want to translate the reduction of a redex $(\lambda P{:}\Delta.A)\,B$, it is enough to reduce first all the redexes $(\lambda y.[\![ A' ]\!])\,[\![ D ]\!]$ appearing in $[\![ (\lambda P{:}\Delta.A)\,B ]\!]$. The reduction of these redexes is not an issue, since the variables $\overline{y}$ are fresh, and the $\rho$-reduction erases the context $\Delta$ from the original term, so it is unnecessary to copy the terms $[\![ D ]\!]$ into the translated term. Then we proceed as in the untyped case.
3 The most important point is to check that our use of postponement variables is correct. We will show the two following properties.

(a) Solvable constraints are not postponed.
Let $(\lambda P.A)\,B$ be a term such that the constraint $P \ll B$ is solvable. Then there exists $\overline{\tau_x}$ such that

$$[\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![ B ]\!] : (\!|P|\!)[\overline{\beta_x := \tau_x}]_{x\in\mathcal{D}om(\Delta)}$$

where $D$ is the (common) type of $P$ and $B$. In particular, this ensures that $(\lambda P.A)\,B$ is not translated as $[\![\,\lambda P.A\,]\!]\,(w\,[\![\,B\,]\!])$.
Indeed, if this constraint is solvable, we have $B = P\theta_{P \ll B}$. According to Lemma 4, we have $[\![\,B\,]\!] = [\![\,P\,]\!][\![\,\theta_{P \ll B}\,]\!][\beta_x := \tau_{x\theta}]$. Theorem 5 shows that
$[\![\Gamma/P; D]\!] \vdash_{\mathsf{F}\omega} [\![\,P\,]\!] : [\![\,D\,]\!]^z[\beta_z := \tau_P] = (\!|P|\!)$.
Moreover we have $[\![\Gamma/P; D]\!] \setminus \overline{\beta_x} \subseteq [\![\Gamma/B; D]\!]$ and

$$[\![\,D\,]\!]^z[\beta_z := \tau_B] = [\![\,D\,]\!]^z[\beta_z := \tau_P][\beta_x := \tau_{x\theta}] = (\!|P|\!)[\beta_x := \tau_{x\theta}]$$

Typing being closed by substitution in $\mathsf{F}\omega$, we can conclude

$$[\![\Gamma/B; D]\!] \vdash_{\mathsf{F}\omega} [\![\,B\,]\!] : (\!|P|\!)[\beta_x := \tau_{x\theta}]$$

(b) The constraints that become solvable appear in the type of the (super-)term where they become solvable.
Let $(\lambda P.A)\,B$ be a $\rho$-term such that the matching constraint $P \ll B$ is not solvable. If $(\lambda P.A)\,B$ is a subterm of a term $A'$ in which the constraint becomes solvable, then in the typing of $A'$ the constraint becomes solvable.
To show that, it is sufficient to consider a kind $\sigma$-long form for types. Let us proceed by induction on the structure of $A'$:

**If $A'$ is the term $A$ itself** then trivially the constraint appears into the type $[P \ll B]C$ of $A$.

**If $A' \equiv A_1 \wr A_2$** then necessarily $A'$ and $A_1$ and $A_2$ have a common type. By induction hypothesis, the constraint $[P \ll B]$ appears in the type of whichever subterm $A_1$ or $A_2$ of which $A$ is a subterm, hence it appears in the type of $A'$.

**If $A' \equiv \lambda P.A_1$** then by induction hypothesis, the constraint appears in the type $D$ of $A_1$, hence it appears in the type $\Pi P.D$ of $A'$.

**If $A' \equiv A_1\,A_2$ where $A$ is a subterm of $A_1$** by inspection of a typing derivation we have $\vdash A_1 : \Pi Q.C'$. The constraint $[P \ll B]$ being unsolvable, $A$ must be a strict subterm of $A_1$; by induction hypothesis, we can take $C'$ such that $C'$ the constraint appears in it, hence it appears in the type $[Q \ll A_2]C'$ of $A'$. Moreover if it is the $\sigma$-reduction of $Q \ll A_2$ which makes the constraint $P \ll B$ solvable, then we have $[Q \ll A_2]C' =_\sigma C'\theta_{Q \ll A_2}$ where the constraint $P \ll B\theta$ becomes solvable.

**If $A' \equiv A_1\,A_2$ where $A$ is a subterm of $A_2$** then by inspection of a typing derivation $\vdash A_1 : \Pi Q.C'$ hence, as $A$ is a subterm of $A_2$, it appears (and so does the constraint) in the type $[Q \ll A_2]C'$ of $A'$.

This last property justifies that the matching constraints that become solvable can be detected during typing, hence during the translation too. The instantiations of postponement variables occurring during the translation of applications are then sufficient, and into the $\lambda$-term $[\![\,A'\,]\!]$ the only postponement variables left correspond to definitive matching failures.

□

**Theorem 6 (Strong normalization of typable $\rho$-terms).**
For all $\Sigma, \Gamma, A, C$, if $\Gamma \vdash_\Sigma A : C$ then $A$ is strongly normalizing.

*Proof.*

**If** $\Gamma \vdash_\Sigma C : *$ then by Theorem 5, we know that there exists $\tau$ such that

$$[\![\Gamma/A; C]\!] \vdash_{\mathsf{F}\omega} [\![ A ]\!] : [\![ C ]\!]^z[\beta_z := \tau].$$

By Lemma 6, if $A$ has an infinite reduction, then $[\![ A ]\!]$ has an infinite reduction. But by strong normalization of System $\mathsf{F}\omega$, the term $[\![ A ]\!]$ has no infinite reduction. Thus, $A$ is strongly normalizing.

**If** $\Gamma \vdash_\Sigma C : \Box$ then $A$ is a $\rho$-type. Considering the shape of types the only reductions which can occur in $A$ are:

— $\sigma$-reductions, whose number is bound by the number of matching constraints appearing into $A$;

— reductions into the terms appearing in $A$, which are finite because of the previous cases.

□

## 8. Strong normalization in the dependent type system

Now we extend the previous result for the $P^2TS$ system with dependent types, *i.e.* allowing the product rules $(*, *)$ and $(*, \Box)$.

To achieve so, we do not need a new target system more powerful than $\mathsf{F}\omega$. Indeed, this proof relies on an encoding of $P^2TS$ with dependent types into $P^2TS$ without dependent types. Thus, we reap immediately the benefits of our strong normalization property for $\rho_\to$ by using it to prove strong normalization for $\rho P$.

**Theorem 7 (Strong normalization in $\rho P$).**
Every $\rho$-term typable in $\rho P$ is strongly normalizing.

*Proof.* We follow the main lines of the proof of strong normalization for LF (Harper et al. 1993). We define a translation $\tau$ of sorts and types, and a translation $|\cdot|$ of $\rho P$-terms and types into $\rho_\to$-terms and types, such that $|\cdot|$ erases all dependent types and preserves reductions. We will use a particular constant 0 such that $\vdash_\Sigma 0 : *$ and a family of constants $\pi_{P:\Delta}$ for each pattern $P$ and each context $\Delta$. The constant $\pi_{P:\Delta}$ has type

$\Pi x_1{:}0 \dots \Pi x_n{:}0.\Pi y{:}(\Pi P{:}\tau(\Delta).0).0$ where $n$ is the number of free variables in $P$.

$$
\begin{array}{rcl}
\varepsilon(\Box) & \triangleq & 0 \\
\varepsilon(*) & \triangleq & 0 \\
\varepsilon(x) & \triangleq & x \qquad \text{if } \vdash_\Sigma x : C : \Box \\
\varepsilon(f) & \triangleq & f \qquad \text{if } \vdash_\Sigma f : C : \Box \\
\varepsilon(\Pi P{:}\Delta.C) & \triangleq & \Pi P{:}\varepsilon(\Delta).\varepsilon(C) \\
\varepsilon(\lambda P{:}\Delta.A) & \triangleq & \varepsilon(A) \\
\varepsilon([P \ll_\Delta B]C) & \triangleq & [P \ll_{\varepsilon(\Delta)} |B|]\varepsilon(C) \\
\varepsilon(A\,B) & \triangleq & \varepsilon(A)
\end{array}
$$

$$
\begin{array}{rcl}
|x| & \triangleq & x \\
|f| & \triangleq & f \\
|\Pi P{:}\Delta.C| & \triangleq & \pi_{P:\Delta}\ |C_1| \dots |C_n|\ (\lambda P{:}\varepsilon(\Delta).|C|) \\
 & & \qquad \text{if } \Delta \equiv x_1{:}C_1 \dots x_n{:}C_n \\
|\lambda P{:}\Delta.A| & \triangleq & \lambda P{:}\varepsilon(\Delta).\big((\lambda y_1{:}0 \dots \lambda y_n{:}0.|A|)\,|C_1| \dots |C_n|\big) \\
 & & \qquad \text{if } \Delta \equiv x_1{:}C_1 \dots x_n{:}C_n \\
|A \wr B| & \triangleq & |A| \wr |B| \\
|A\,B| & \triangleq & |A|\,|B| \\
|[P \ll_\Delta B]C| & \triangleq & [P \ll_{\varepsilon(\Delta)} |B|]\ \big((\lambda y_1{:}0 \dots \lambda y_n{:}0.|C|)\,|C_1| \dots |C_n|\big) \\
 & & \qquad \text{if } \Delta \equiv x_1{:}C_1 \dots x_n{:}C_n
\end{array}
$$

The function $\varepsilon$ is extended to contexts and signatures, where it operates over each type. The correctness of these functions is ensured by the three following lemmas. □

**Lemma 7.**
If $\Gamma \vdash_\Sigma B : C : \Box$ or $\Gamma \vdash_\Sigma B : \Box$ in $\rho P$, then $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon(B) : *$ in $\rho_\to$.

*Proof.* By induction on $B$. Immediate if $\vdash_\Sigma B : \Box$, the only interesting cases for $\vdash_\Sigma B : C : \Box$ are abstraction and application:

**If** $B \equiv \lambda P{:}\Delta.B_1$ then $C =_{\beta\delta} \Pi P{:}\Delta.C_1$ with $\Gamma, \Delta \vdash_\Sigma B_1 : C_1$ and $\Gamma \vdash_\Sigma \Pi P{:}\Delta.C_1 : \Box$, hence $\Gamma, \Delta \vdash_\Sigma C_1 : \Box$.
By induction hypothesis $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} \varepsilon(B_1) : *$, hence the same is true for $\lambda P{:}\Delta.B_1$.

**If** $B \equiv B_1\,B_2$ then $C =_{\beta\delta} [P \ll B_2]C_1$ with $\Gamma \vdash_\Sigma B_1 : \Pi P.C_1$ and $\Gamma \vdash_\Sigma [P \ll B_2]C_1 : \Box$.
Then we have $\Gamma \vdash_\Sigma C_1 : \Box$, hence $\Gamma \vdash_\Sigma B_1 : \Pi P.C_1 : \Box$; by induction hypothesis $\varepsilon(\Gamma) \vdash_\Sigma \varepsilon(B_1\,B_2) = \varepsilon(B_1) : *$.

□

**Lemma 8.**
If $\Gamma \vdash_\Sigma A : C$ in $\rho P$, then $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |A| : \varepsilon(C)$ in $\rho_\to$.

*Proof.* By induction on a derivation of $\Gamma \vdash_\Sigma A : C$, distinguishing over the last used rule.

**If the last rule is** (Sig), (Weak$\Sigma$) **or** (Axiom) it is immediate. For every premise $\vdash_\Sigma C : s$, according to Lemma 7 we have $\vdash_{\varepsilon(\Sigma)} \varepsilon(C) : *$.

**If the last rule is**

$$(\textsc{Var})\ \frac{\Gamma \vdash_\Sigma C : s}{\Gamma, x{:}C \vdash_\Sigma x : C}$$

By Lemma 7 we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon(C) : *$. As $\varepsilon(\Gamma, x{:}C)(x) = \varepsilon(C)$ we have $\varepsilon(\Gamma, x{:}C) \vdash_{\varepsilon(\Sigma)} x : \varepsilon(C)$.

**If the last rule is**

$$(\textsc{Const})\ \frac{\Sigma\ sig \qquad f : C \in \Sigma}{\vdash_\Sigma f : C}$$

Immediately $\varepsilon(\Sigma)(f) = \varepsilon(C)$ and by Lemma 7 we have $\vdash_{\varepsilon(\Sigma)} \varepsilon(C) : *$.

**If the last rule is** (WeakΓ) it is immediate: $\varepsilon(\Gamma, x{:}C) = \varepsilon(\Gamma), x{:}\varepsilon(C)$ and by Lemma 7 we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon(C) : *$.

**If the last rule is** (Conv) it is easy to see that if $C =_{\rho\delta} B$, then $\varepsilon(C) =_{\rho\delta} \varepsilon(B)$, and by Lemma 7 we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon(C) : *$. We can apply the induction hypothesis and a conversion step in $\rho_\to$.

**If the last rule is**

$$(\textsc{Struct})\ \frac{\Gamma \vdash_\Sigma A : C \qquad \Gamma \vdash_\Sigma B : C}{\Gamma \vdash_\Sigma A \wr B : C}$$

By induction hypothesis $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |A| : \varepsilon(C)$ and $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |B| : \varepsilon(C)$, hence $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |A \wr B| : \varepsilon(C)$

**If the last rule is**

$$(\textsc{Abs})\ \frac{\Gamma, \Delta \vdash_\Sigma A : C \qquad \Gamma \vdash_\Sigma \Pi P{:}\Delta.C : s}{\Gamma \vdash_\Sigma \lambda P{:}\Delta.A : \Pi P{:}\Delta.C}$$

By induction hypothesis $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |A| : \varepsilon(C)$ and by Lemma 7 we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon(\Pi P{:}\Delta.C) : *$.

Moreover, $|\lambda P{:}\Delta.A| = \lambda P{:}\varepsilon(\Delta).\big((\lambda y_1{:}0 \ldots \lambda y_n{:}0.|B|)\,|C_1| \ldots |C_n|\big)$, where the $n$ arguments $|C_i|$ are absorbed by the $n$ abstractions over the $y_i$, and do not influence typing (by induction hypothesis we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |C_i| : \varepsilon(s) = 0$ which is the expected type for $y_i$).

Indeed $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |\lambda P{:}\Delta.A| : \Pi P{:}\varepsilon(\Delta).\varepsilon(C) = \varepsilon(\Pi P{:}\Delta.C)$.

**If the last rule is**

$$(\textsc{Appl})\ \frac{\Gamma \vdash_\Sigma A : \Pi P{:}\Delta.C \qquad \Gamma \vdash_\Sigma [P \ll_\Delta B]C : s}{\Gamma \vdash_\Sigma A\,B : [P \ll_\Delta B]C}$$

By induction hypothesis, $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |A| : \varepsilon(\Pi P{:}\Delta.C) = \Pi P{:}\varepsilon(\Delta).\varepsilon(C)$ and by Lemma 7 we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} \varepsilon([P \ll_\Delta B]C) : *$, hence $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |A\,B| : [P \ll_{\varepsilon(\Delta)} |B|]\varepsilon(C) = \varepsilon([P \ll_\Delta B]C)$.

**If the last rule is**

$$(\textsc{Prod})\ \frac{\Gamma, \Delta \vdash_\Sigma P : A \qquad \Gamma, \Delta \vdash_\Sigma A : s_1 \qquad \Gamma, \Delta \vdash_\Sigma C : s_2}{\Gamma \vdash_\Sigma \Pi P{:}\Delta.C : s_2}$$

By induction hypothesis $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |C| : \varepsilon(s_2) = 0$ and for every $(x_i{:}C_i) \in \Delta$ we

have $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |C_i| : \varepsilon(s) = 0$. Given the type of $\pi_{P:\Delta}$, we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |\Pi P{:}\Delta.C| : 0 = \varepsilon(s_3)$.

**If the last rule is**

$$(\textsc{Match})\ \frac{\Gamma, \Delta \vdash_\Sigma P : A \quad \Gamma \vdash_\Sigma B : A \quad \Gamma, \Delta \vdash_\Sigma A : s_1 \quad \Gamma, \Delta \vdash_\Sigma C : s_2}{\Gamma \vdash_\Sigma [P \ll_\Delta B]C : s_2}$$

Notice that $|P| = P$. By induction hypothesis $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} P : \varepsilon(A)$ and $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |B| : \varepsilon(A)$ and $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |C| : \varepsilon(s_2) = 0$.
Still by induction hypothesis, for every $(x_i{:}C_i) \in \Delta$ we have $\varepsilon(\Gamma, \Delta) \vdash_{\varepsilon(\Sigma)} |C_i| : \varepsilon(s) = 0$ which correspond indeed to the type expected by the $y_i$.
Thus, we have $\varepsilon(\Gamma) \vdash_{\varepsilon(\Sigma)} |[P \ll_\Delta B]C| : 0 = \varepsilon(s_3)$.

□

**Lemma 9.**
If $A \mapsto_{\rho\sigma\delta} A'$, then $|A| \mapsto\!\!\!\!\to_{\rho\sigma\delta} |A'|$ in at least one step.

*Proof.* Immediate: for each term $A$, for whichever subterm $B$ that can be reduced (even if $B$ appears in a type), the term $|B|$ appears in $|A|$. Moreover, the $\rho$ (resp. $\delta$)-redexes are translated by $\rho$ (resp. $\delta$)-redexes. □

## 9. Conclusion and perspectives

In this paper, we have proved strong normalization of the simply-typed and dependently-typed $P^2TS$. The proof relies on a faithful translation from simply-typed $P^2TS$ into System $\mathsf{F}\omega$, and then from dependently-typed $P^2TS$ into simply-typed $P^2TS$.

In the untyped framework, we encoded pattern matching in the $\lambda$-calculus in a quite efficient way, ensuring that every $\rho\sigma\delta$-reduction is translated into (at least) one $\beta$-reduction. Introducing types in the translation proved an interesting challenge. One difficulty comes from the pattern matching occurring in the $P^2TS$ types, which calls for accurate adjustments in the translation. Another remarkable point is that the typing mechanisms of even the simply-typed $P^2TS$ can be expressed only with the expressive power of System $\mathsf{F}\omega$, which is rather surprising since $\mathsf{F}\omega$ is a higher-order system featuring types depending on types.

An interesting development of this work would be to adapt the proof for the other type systems of $P^2TS$. Two work directions are possible. On the one hand, we can adapt the typed translation for typing judgments in the more complex type systems. On the other hand, we can expect to find (typed) translations from some type systems of $P^2TS$ into other ones, as we have done for dependent types.

It would also be interesting to devise a model-theoretic proof of strong normalization for $P^2TS$: an interpretation of types as functional spaces is generally the first step towards their interpretation as propositions, which is a key to the definition of a Curry-Howard isomorphism. The encoding presented here could provide a basis for this interpretation: all the translated contexts contain the hypothesis $x_\perp : \perp$ (*falsum*). However, if we manage

to characterize which $\lambda$-terms produced by the encoding do not use the assumption $\bot$, we already have a suitable interpretation for the corresponding $P^2TS$ terms.

A third research direction is to focus on the (Struct) rule: it seems that, with intersection types, we could obtain an original treatment of the conjunction connector, where structures correspond to introduction and matching failures enable elimination.

In the long term, we expect to use $P^2TS$ as the base language for a powerful proof assistant combining the logical soundness of the $\lambda$-calculus and the computational power of the rewriting. This proof of strong normalization is a main stepstone for this research direction, since logical soundness is deeply related to strong normalization.

**Acknowledgments**
The authors would like to thank Horatiu Cirstea, Claude Kirchner and Luigi Liquori for the constant support and interest they put in this work; Patrick Blackburn for some useful insights about the expressiveness of the typed $\lambda$-calculus; Sylvain Salvati for many fruitful informal discussions about System F; Frédéric Blanqui, Gilles Dowek and the anonymous referees for their valuable comments.

## References

G. Dowek, T. Hardin, C. Kirchner, Theorem proving modulo, Journal of Automated Reasoning 31 (1) (2003) 33–72.

B. Werner, *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université Paris 7, 1994.

J. Klop, V. van Oostrom, F. van Raamsdonk, Combinatory reduction systems: introduction and survey, Theoretical Computer Science 121 (1993) 279–308.

T. Nipkow, C. Prehofer, Higher-order rewriting and equational reasoning, in: W. Bibel, P. Schmitt (Eds.), Automated Deduction — A Basis for Applications. Volume I: Foundations, Kluwer, 1998.

T. Coquand, Pattern matching with dependent types, in: Informal proceedings workshop on types for proofs and programs, Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., Båstad, Suède, 1992, pp. 71 – 84.

D. Kesner, L. Puel and V. Tannen, A typed pattern calculus, *Information and Computation*, Vol. 124 (1), 1996, pp. 32–61.

F. Blanqui, Definitions by rewriting in the calculus of constructions, in: LICS, 2001, pp. 9–18.

F. Blanqui, J.-P. Jouannaud, M. Okada, Inductive-data-type systems, TCS 272 (1–2) (2002) 41–68.

H. Cirstea, C. Kirchner, L. Liquori, The Rho Cube, in: F. Honsell (Ed.), Foundations of Software Science and Computation Structures, Vol. 2030 of Lecture Notes in Computer Science, Genova, Italy, 2001, pp. 166–180.

G. Barthe, H. Cirstea, C. Kirchner, L. Liquori, Pure Patterns Type Systems, in: Principles of Programming Languages - POPL2003, New Orleans, USA, ACM, 2003.

H. Cirstea, L. Liquori, B. Wack, Rewriting calculus with fixpoints: Untyped and first-order systems, in S. Berardi, M. Coppo, F. Damiani, editors, *International Workshop on Types for Proofs and Programs, TYPES 2003*, Vol. 3085 of *Lecture Notes in Computer Science*, pp. 147–161, Torino, Italy, 2003. Springer.

H. P. Barendregt, Lambda calculi with types, in S. Abramsky, D. Gabbay, T. Maibaum, editors, *Handbook of Logic in Computer Science*. Clarendon Press, 1992.

B. Wack, The simply-typed pure pattern type system ensures strong normalization, in J.-J. Lévy, E. Mayr and J. Mitchell, editors, *Third International Conference on Theoretical Computer Science, IFIP TCS'04*, pp. 633 – 646, Toulouse, France, August 2004. IFIP, Kluwer Academic Publishers.

H. Cirstea, C. Kirchner, The typed rewriting calculus, in: Third International Workshop on Rewriting Logic and Application, Kanazawa (Japan), 2000.

H. Cirstea, C. Kirchner, L. Liquori, B. Wack, The rewriting calculus : some results, some problems, in: D. Kesner, F. van Raamsdonk, T. Nipkow (Eds.), The first international workshop on Higher-Order Rewriting, FLoC'02, Copenhagen, Denmark, 2002, lORIA Internal research report A02-R-470.

J.-Y. Girard, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat, Université Paris VII, juin 1972.

Th. Coquand, G. Huet, The calculus of constructions, *Information and Computation*, vol. 76, 1988, pp. 95 – 120.

C. Kirchner, F. Klay, Syntactic theories and unification, in: Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA), 1990, pp. 270–277.

R. Harper, F. Honsell, G. Plotkin, A framework for defining logics, Journal of the ACM.